MASTER THESIS

Michal Zdražil

From computer 3D modelling to reality and back

Department of Mathematics Education

Supervisor of the master thesis:	RNDr. Petra Surynková, Ph.D.
Study programme:	Mathematics
Study branch:	Mathematics education - Descriptive geometry education

Prague 2021

Above all others, I would like to thank my supervisor, who was supporting, infinitely patient and always helpful, when I could not find the way forward and was far far from the diligent hardworking student I should have been. I am also thankful to my friends who supported me morally and supplied me with coffee and cakes in times of struggle.

Title: From computer 3D modelling to reality and back

Author: Michal Zdražil

Department: Department of Mathematics Education

Supervisor: RNDr. Petra Surynková, Ph.D., Department of Mathematics Education

Abstract: Technological advancements are faster than ever and on the frontier are applications and mechanisms entwined with 3D computer aided modelling, such as 3D printing, scanning and extended reality technologies. This work gives a peek behind the veil of mystery surrounding these technologies. We aim to give a brief look into each of the mentioned areas and let the reader experience them practically, mathematically and algorithmically in hope to bring these three so often separated views closer together and to the reader.

Keywords: Regular surface, modelling, projection, photogrammetry, 3D printing, surface reconstruction, augmented reality

Contents

Introduction

Since its modest beginnings in the late 1960s, computer-aided design (CAD) and modelling have established their place among other offspring of mathematics and IT and have become the daily bread of an uncountable number of industries and creative fields as a means of representing three-dimensional objects. The emergence of new technologies entwined with CAD, such as 3D printing, 3D scanning, virtual reality (VR) and augmented reality (AR), and their ever-growing applicability has even increased the demand for not only proficient CAD users but also capable developers of CAD software.

The most common representation of three-dimensional objects in CAD is the boundary representation (B-Rep) which reduces an object to a collection of (preferably) regular surfaces, that is without spikes or sharp edges, basically turning a solid into a hollow shell. Therefore the spotlight of CAD is on representations of surfaces (the parametric representation of which currently turns out to be the most prominent) rather than solids in the volume sense, only rarely do we encounter an actual volume representation despite its undeniable usefulness in certain areas (such as tomography).

In most cases a design project naturally does not start with the final object itself but rather with an idea that is meant to be turned into one. (A reverse process often used in architecture, for example, is to scan an existing physical model which is then adjusted using computer software.) Then comes the time for a CAD software. In CAD software, primitives - be they points, curves or representations of basic 3D objects - can be manipulated by affine and projective transformations or combined by Boolean operations in order to obtain the desired model. Such a model can further be restructured and interpreted for purposes other than it had initially.

While the process of creating, representing and displaying three- (or higher-) dimensional surfaces on computer screens has been well documented, many fields of its applications still contain gaps to be filled and paths to be discovered, such as more efficient ways of triangulating and quadrangulating, higher quality real-time rendering, numerical optimization, rationalization, more user friendly modelling software which could help satisfy the thirst for more workers qualified to work with them and many more mostly a specific industry-related problems.

This thesis is a survey attempting to offer views from the three possible perspectives vital for a fuller understanding of CAD in theory and practice – one that cannot be achieved from one of these perspectives alone. These views are those of a CAD user (via showing the modelling process step by step on specific examples in a modelling software leading all the way to 3D printing and basics of VR and AR), a CAD developer (via creating a simple software for surface projections) and an expert in computational geometry (developments in which field have implications for CAD software capabilities). By giving a self-contained account of these views together, it is hoped that the benefits of engaging with them all are evident – that deepening the interest of a practitioner, developer or theoretician in the work of the other two will bring about great strides in problem-solving capabilities.

Let us start with a rendered image (fig. 1). How was it made? How is it displayed? Can it be used further? 3D printed perhaps? How?

On the following pages we will try to walk through a process which led to this image and further possible avenues, get a taste of the underlying mathematics and algorithms. We hope that through this will help those interested in computer graphics (from any point of view) to overcome the first steps and find new sources of information. For the scope of the work, each chapter is kept reasonably short so a lot of information is omitted, therefore there are always some additional sources mentioned for deeper studying of the topic.

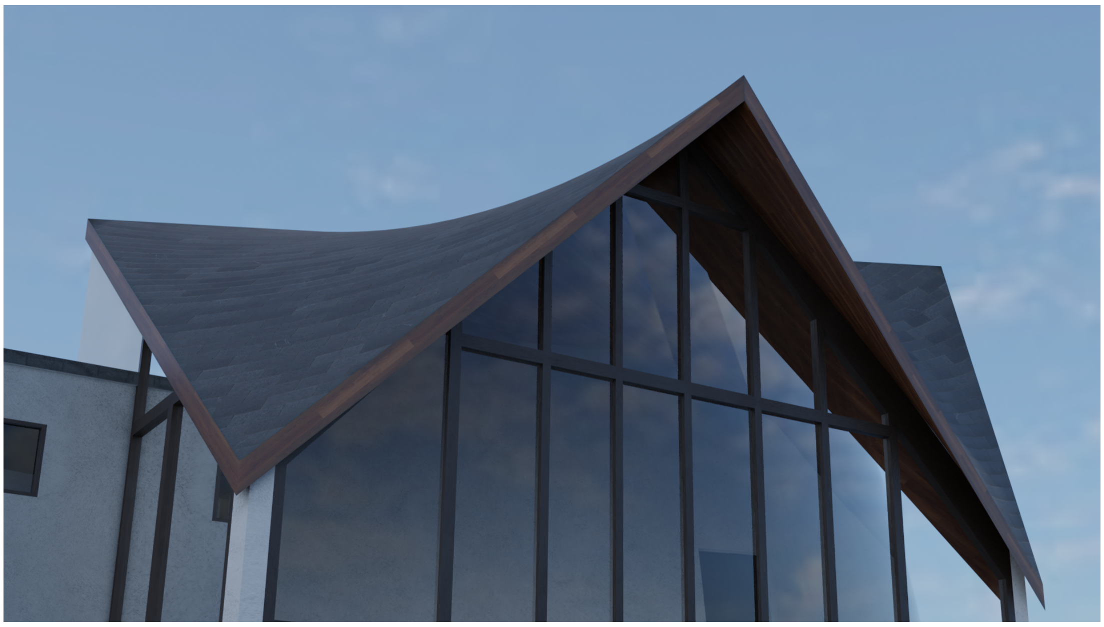


Figure 1: There is a long way from idea to an actual render, steps we will try to take in this thesis.

In the first chapter a quick peek into surface representations (using computer or not) is offered.

Second chapter deals with computer aided modelling itself – methods for modelling via a CAD program with an extra portion of mathematics is presented followed by a simplified pipeline of displaying 3D scene on one's monitor.

The third chapter presents in a way opposite attitude to modelling – retrieving an object from real-world data, namely photographs.

Having obtained an object in one way or another, we might want to see it outside of a computer screen too. 3D printing is a way to do that. We walk through its methods in the fourth chapter.

No 3D printer? No problem! A mere mobile phone might be enough to see a model in the world around. Augmented reality is the way in this case as we describe in the fifth chapter.

The last chapter presents chosen parts of a program for modelling NURBS surfaces. It is very simplified and basic (possibly because the author is also a beginner programmer), but aims to shed light on some basic concepts of algoritmization of selected computer graphics tools presented above.

1. Analytic representations of surfaces

There are three main ways used to describe and express surfaces in a three-dimensional space. Those are namely parametric, implicit and explicit methods, each comes with its respective advantages and disadvantages. All these aim to express points of the desired surface in terms of (x, y, z) coordinates in their own ways. [1]

The *parametric* representation expresses individual coordinates of a point (x, y, z) as functions of the parameters u and v in closed intervals:

$$x = x(u, v), y = y(u, v), z = z(u, v),$$

where $u_1 \leq u \leq u_2$ and $v_1 \leq v \leq v_2$. Naturally, there are some requirements on the functions that have to be met in order to consider the given representation a surface *patch*. The functions $x(u, v)$, $y(u, v)$ and $z(u, v)$ should be continuous and have the desired number of continuous partial derivatives. The parametric surface is said to be of *class* c, if all the given functions have continuous partial derivatives up to the order c. In the case the class is not explicitly given, it is supposed to be the case that the functions have infinitely many continuous derivatives.

Using the vector notation the parametric surface can be expressed by a vector-valued function

$$\vec{r} = \vec{r}(u, v).$$

This representation allows for expressing a large variety of surfaces – closed, multivalued, with infinite slopes etc. It is also independent of the system of axes, therefore it is easy to transform surfaces and fit and manipulate free-form shapes using the parametric representation. The high flexibility of this representation can sometimes mess up with the intersections and point classification.

The *implicit* representation of a surface defines the locus of points (x, y, z) which satisfy an equation of the form

$$f(x, y, z) = 0. \tag{1.1}$$

The degree of (1.1) can tell us lot about the surface by itself. If it is linear, it implies that the surface is a plane, if it is of the second degree, the surface is a *quadric* and so on.

This representation allows for a similar variety of shapes to the parametric one, but one of its big disadvantages is the dependence on the axes which makes it difficult to transform specific surfaces and causes the free-form shapes to be difficult to manipulate. On the other hand the point classification (with respect to the surface) is easy which simplifies for instance solid modelling.

The *explicit* form can be derived from the previous two if certain conditions hold. If the implicit equation (1.1) can be solved for one of the variables as a function of the other two, let us suppose z is solved in terms of x and y, we obtain an explicit surface

$$z = F(x, y). \tag{1.2}$$

This is always at least locally possible if $\frac{\partial f}{\partial z} \neq 0$ [2].

In case that the variables u and v of the parametric form can be solved in terms of x and y as $u = y(x, y)$ and $v = v(x, y)$, we may substitute these into $z = z(u, v)$ which yields the explicit form of a given surface. This is possible if and only if $\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u} \neq 0$ [2].

Conversely the implicit form can easily be obtained from the explicit one by subtracting $F(x, y)$ from both sides of the equation (1.2) and the parametric one can be derived by setting $x = u$, $y = v$ and $z = F(u, v)$. Therefore we consider the explicit form a special case of the previous two.

When it comes to the gamut of surfaces the explicit form allows, it does not provide us with as many options as the other two. It is also axes dependent which comes with the burdens mentioned above. All that makes it the least viable option when it comes to the surface representation.

For all practical purposes, not only by the optics of the computed-aided geometry, the following text will be focused almost exclusively on the parametric representations.

Before we dive deeper into the world of parametric representations, let us first consider a couple examples that aim to shed light on the relations between individual ways of expressing surfaces:

Example 1. *Let us consider a hyperbolic paraboloid patch (fig. 1.1) given by a parametric representation:*

$$r(u, v) = \begin{bmatrix} u + v \\ u - v \\ u^2 - v^2 \end{bmatrix}; u, v \in \langle 0; 1\rangle \, .$$

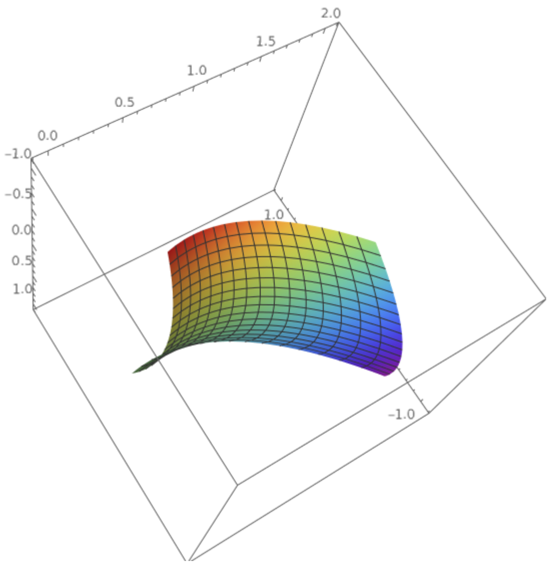


Figure 1.1: A hyperbolic paraboloid unit patch.

From here $x = u + v$, $y = u - v$, $z = u^2 - v^2$ and it is easy to see that u and v can be expressed in terms of x and y as $u = \frac{x+y}{2}$ and $v = \frac{x-y}{2}$. From here

$z = \frac{x^2+2xy+y^2}{4} - \frac{x^2-2xy+y^2}{4} = xy$ for $x+y, x-y \in \langle 0;2 \rangle$ is the explicit form and from here $z - xy = 0$ is the implicit form.

Example 2. *Consider a unit sphere (fig. 1.2) given by a parametric representation:*

$$r(\theta,\phi) = \begin{bmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{bmatrix}; \theta \in \langle 0;\pi \rangle, \phi \in \langle 0; 2\pi \rangle.$$

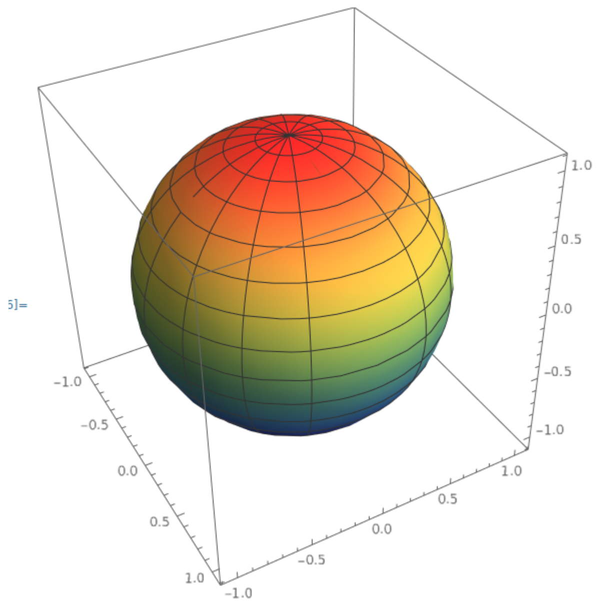


Figure 1.2: Unit sphere.

A sphere can not be represented by a single explicit surface, therefore even looking for the implicit one will be a little harder than in the example above. First, we express θ in terms of z, specifically $\theta = \arccos z$. Now, we can plug this expression into the equations for x and y after applying the fact that $\sin(\arccos z) = \sqrt{1-z^2}$ for all $z \in \langle -1; 1 \rangle$ we obtain the following equations $x = \sqrt{1-z^2}\cos\phi$ and $y = \sqrt{1-z^2}\sin\phi$. Now we can square both equations, sum them and after applying the equality $\sin^2\phi + \cos^2\phi = 1$ we obtain the familiar expression $x^2 + y^2 = 1 - z^2 \Rightarrow x^2 + y^2 + z^2 = 1$, which is an implicit representation of the unit sphere. It isn't hard to verify that all the steps above were equivalences for the given intervals and that this equation can not be rewritten in a single explicit form for any of the variables.

1.1 Surfaces in computer aided modelling

This part of the thesis introduces certain techniques of and approaches to computer aided modelling and representations of three-dimensional objects.

Modern-day computer graphic software represents 3D models in various ways. The most common one and the one we will focus on the most in the following text would be the boundary representation (B-rep) – depending on the use we can often encounter analytical models (using Bézier, B-spline or more specifically NURBS surfaces or eventually even implicit surfaces) and polygonal models, sometimes referred to as *meshes* (used mostly in time-sensitive applications). The difference between analytical surfaces and meshes is obvious – smoothness (fig. 1.3)

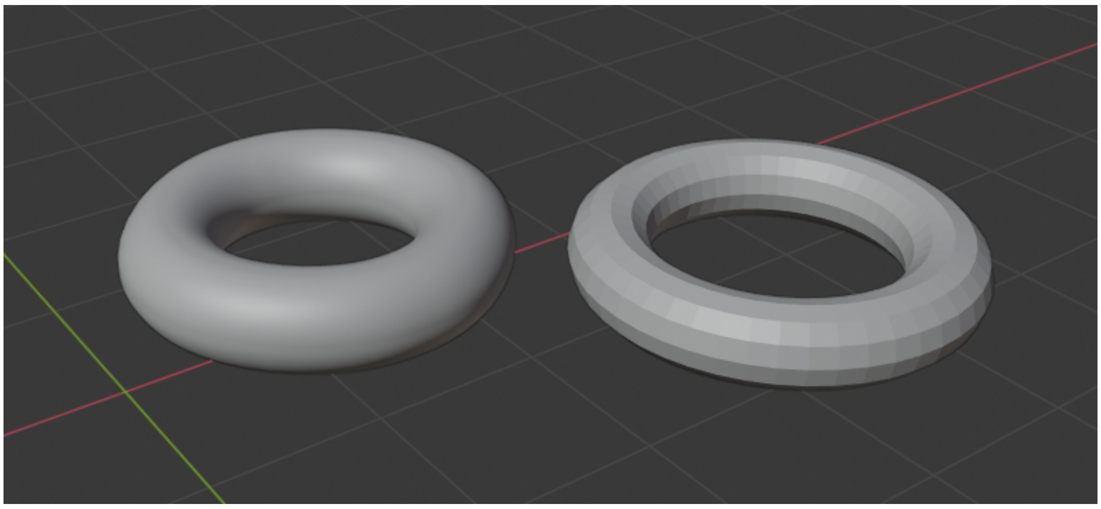


Figure 1.3: A NURBS surface (left) and a mesh (right) in a modelling program.

The less usual type of representation, which is utilized mostly when a certain object or data set can't be easily described in geometrical terms or carries additional information, is the volume representation. A good example of this representation's use in practice is creating 3D models using tomography.

Bézier, B-spline and the best of both worlds

Various modelling programs tackle the modelling process in various ways depending on the use of the final model. As a part of most of their interfaces, the user is provided with a large scale of modelling tools from which one can choose depending on their preference. When it comes to modelling which requires high precision and robust mathematical description, such as the modelling for the purposes of the automotive (which was actually the driving force in the development of this area of modelling methods), architecture or various other fields of industry, the software providing the user with the most suitable environments is usually the CAD (Computer Aided Design) software.

The means by which CAD software achieves this is the use of the surfaces presented in this section which are not only very intuitive and easy to manipulate, do not require any mathematical background from the user, but are also relatively easy to describe, compute and visualize for the software itself. Namely these are the Bézier and B-spline surfaces and their in today's 3D modelling unrivalled offspring – the NURBS surfaces.

1.1.1 Bézier surfaces

Bézier surfaces were there in the early days of computer aided 3D modelling and represent a mathematical apparatus made specifically for the use of CAD. Even though their use in their pure form is in decline, they still act as a very good entry point to the world of analytical representations of surfaces and will help us understand the functionality of more complex surface descriptions.

Definition 1 (Bézier surface)**.** *[3] A Cartesian or tensor product surface patch $Q(u,v)$ of degree $m \times n$ is called the Bézier surface if it satisfies the following equation:*

$$Q(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} A_{i,j} B_{i,m}(u) B_{j,n}(v); \quad 0 \le u, v \le 1, \tag{1.3}$$

where $A_{i,j}$ are the control points which together with a set of straight lines connecting the consecutive ones form the surface's control grid and $B_{i,m}$, $B_{j,n}$ are the Bernstein basis functions of degrees m and n in the u and v parametric directions given by:

$$B_{i,m}(u) = \frac{m!}{i!(m-i)!} u^i (1-u)^{m-i}$$

$$B_{j,n}(v) = \frac{n!}{j!(n-j)!} v^j (1-v)^{n-j},$$

where $0^0 \equiv 1$.

As the name suggests *the control points* of the $(m+1) \times (n+1)$ grid help us control the shape of the surface. After a closer look we can see that all of the points of the grid generally do not have to coincide with the surface – it is an approximation surface [4].

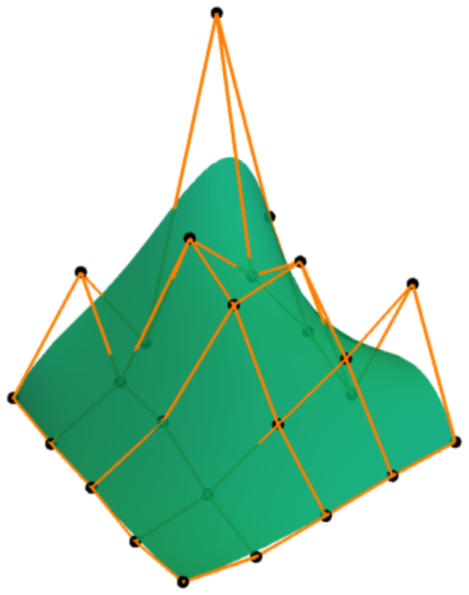


Figure 1.4: An example Bézier patch and its control net.

If we take a step forward and look at the equation (and fig. 1.4) we can derive some basic properties of these patches [5]:

- The corner points of the control grid are always contained in the Bézier surface.
- The surface generally follows the shape of the control grid.

- The surface is contained within the convex hull of the control grid.
- The surface is invariant under affine transformations.

These all are very advantageous properties for computational purposes which also make the use of the said surface very comfortable for the user. There are two major pitfalls however:

- A change of a single point of the control grid has a global impact on the shape of the surface.
- The degree of the surface is directly linked to the number of control points.

For these reasons the usual practise is joining several Bézier patches in order to create the desired surface via a process called *patching* (fig.1.5).

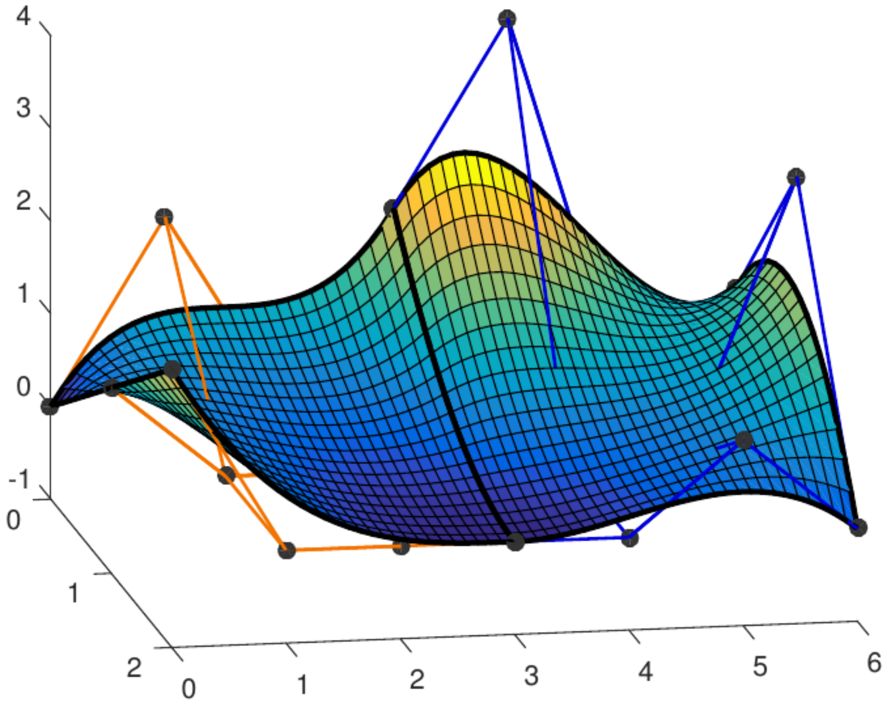


Figure 1.5: Two Bézier surfaces patched together with C^1 continuity. This is achieved by matching the boundary curve (C^0), direction (G^1) and length (C^1) of its derivatives along this curve. (Courtesy Chudáčková [6])

1.1.2 B-spline surfaces

Addressing the latter problems of the Bézier surfaces, the non-rational B-spline (*basis spline*) surfaces emerged. Whilst maintaining some of the important properties of its predecessor, the B-spline and especially its non-uniform rational counterpart offer a way larger scale of possible shapes, localize the changes caused by changes in the control grid and allow for better smoothness control.

Definition 2 (B-spline surface). *A Cartesian or tensor product surface (patch) $Q(u, v)$ is called the B-spline surface if it satisfies the following equation [4]:*

$$\begin{aligned} &Q(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} A_{i,j} N_{i,k}(u) N_{j,l}(v); \\ &m+1 \geq k \geq 1, n+1 \geq l \geq 1, \\ &u \in [u_{k-1}, u_{m+1}], v \in [v_{l-1}, v_{n+1}], \end{aligned} \tag{1.4}$$

where $A_{i,j}$ are its control points forming a topologically rectangular set and $N_{i,k}$, $N_{j,l}$ are B-spline basis functions of degrees k and l in the directions u and v defined as:

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1 & \textit{if } u_i \leq u < u_{i+1} \\ 0 & \textit{otherwise} \end{cases} \\ N_{i,k}(u) &= \frac{(u-u_i)N_{i,k}(u)}{u_{i+k}-u_i} + \frac{(u_{i+k+1}-u)N_{i+1,k-1}(u)}{u_{i+k+1}-u_{i+1}} \\ \textit{and} \qquad N_{j,0}(v) &= \begin{cases} 1 & \textit{if } v_j \leq v < v_{j+1} \\ 0 & \textit{otherwise} \end{cases} \\ N_{j,l}(v) &= \frac{(v-v_j)N_{j,l}(v)}{v_{j+l}-v_j} + \frac{(v_{j+l+1}-v)N_{j+1,l-1}(v)}{v_{j+l+1}-v_{j+1}}, \end{aligned}$$

where u_i and v_j are the elements (knots) of knot vectors[1]*.*

It is important to note that we do not use the whole knot vector span for the final surface construction. Some parts are only used for computations, they remain passive, because there, the surface no longer behaves as nice as we would like. Specifically this happens for spans where $i < k$ or $p - i < k$, where p is the number of knot spans and k the degree of the basis function.

Let us now shed some light on the concept of knot vectors and their influence on the final shape of the surface. Half-opened intervals between two knots are called *knot spans.* The surface is generated by letting the parameters run through these intervals which forms the basis functions. The significance of the knot vectors is shown in figure 1.6 where two surfaces with the same control net and degree but different knot vectors can be seen. From the definition of knot vectors, the length of a knot span can be equal to zero. We use this property to change and accent the influence of certain parts of the control net.

It is worth noting that each B-spline basis function of k-degree is obtained as a linear combination of two $(k-1)$-degree functions. This means that in order to obtain a set of k-degree basis functions with $m+1$ given points in a direction, we require $m+k+1$ knot spans, that is $m+k+2$ knots.

A curious reader might wonder where the connection between the Bézier and B-spline surfaces lies. It is closely related to the knot vectors. It can be observed that if we set $k = m$, $u_1 = \ldots = u_{n+1} = 0$ and $u_{n+1} = \ldots = n_{2n+2} = 1$, the B-spline basis functions restricted to the interval $[0, 1]$ reduce to Bernstein basis functions. Knot vectors are henceforth the advantage that the B-spline has over Bézier surfaces which allows for the wider scale of shapes.

[1] $\vec{u} = (u_0, \ldots, u_i, u_{i+1}, \ldots, u_p) : \forall i \in \{0, \ldots, p\} : u_i \leq u_{i+1}$

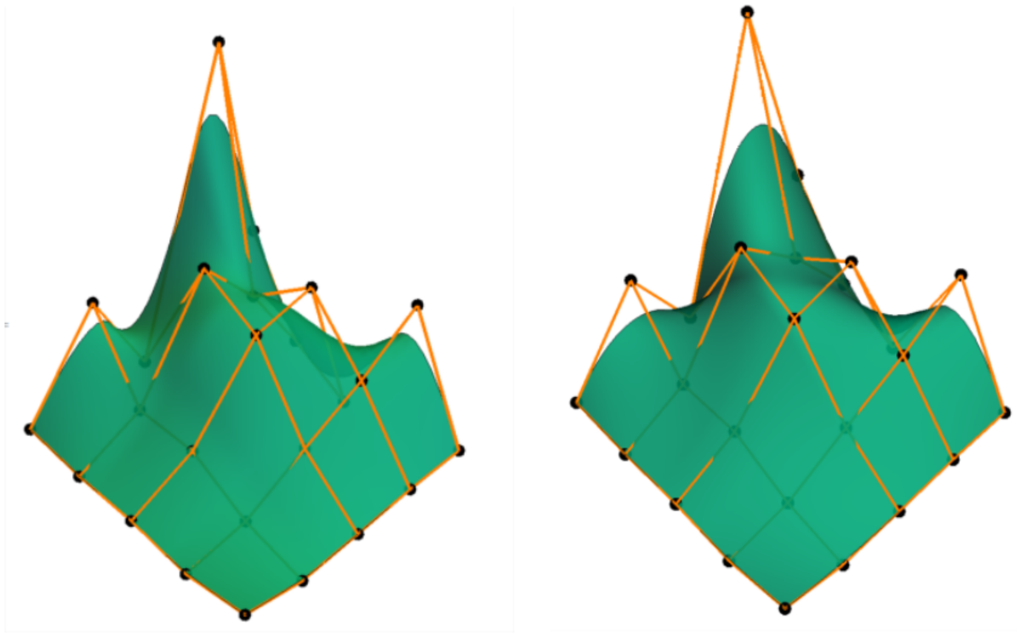


Figure 1.6: Two B-spline patches with the same control net differing in the value one knot in each parametric direction.

As mentioned and partially shown above, B-spline surfaces hold much of the properties of Bézier surfaces (the property of corner points does not generally hold but can be achieved either by higher control point multiplicity or by appropriately setting the knot vectors) while keeping changes caused by adjusting a point of the control grid local. That in addition to easing the user-surface interaction, saving some computation time and increasing the surface shape variety also allows for a more efficient plating with higher connectivity. A new patch can be attached to an existing one by using $(m+1) \times n$ points of the former patch and adding another $m+1$ points – like this we can grant C^{n-1} connectivity.

1.1.3 NURBS surfaces

By further generalizing the B-spline surfaces, we can come all the way to the golden standard of modern computer aided modelling - the NURBS (*non uniform rational B-spline*) surfaces. These allow for a new wide scale of surfaces including free-form surfaces, ruled surfaces and many more. All these objects are represented in terms of the same data structure and algorithms and therefore open the door to a uniform means of interface and processing.

Definition 3 (NURBS surface)**.** *[4] A Cartesian or tensor product surface patch $Q(u,v)$ is called the NURBS surface if it satisfies the following equation:*

$$\begin{aligned} &Q(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j}A_{i,j}N_{i,k}(u)N_{i,l}(v)}{\sum_{i=0}^{n}\sum_{j=0}^{m} w_{i,j}N_{i,k}(u)N_{i,l}(v)}, \\ &m \geq k \geq 0, n+ \geq l \geq 0, \\ &u \in [u_k, u_{m+1}], v \in [v_l, v_{n+1}], \end{aligned} \tag{1.5}$$

where $A_{i,j}$ are once again the control net vertices, $w_{i,j}$ is the weighting factor of the point $A_{i,j}$ and $N_{i,k}$, $N_{j,l}$ are B-spline basis functions as established in the definition of the B-spline surface.

If we choose so, we can rewrite the equation 1.5 using so-called *rational B-spline basis*

$$R_{i,k}(t) = \frac{w_i N_{i,k}(t)}{\sum_{\hat{i}=0}^{n} w_{\hat{i}} N_{\hat{i},k}(t)}$$

into a form more reminiscent of both Bézier and B-spline surfaces [7]:

$$Q(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} A_{i,j} R_{i,k}(u) R_{j,l}(v).$$

From this point, it actually is not too hard to see why the NURBS is a generalization of both the latter. If we set the weight $w_{i,j} = 1$ for all i, j, the B-spline surface is recovered. Furthermore, if it holds that the number of control points is equal to the order of the B-spline basis functions in both directions u and v, then the NURBS surface reduces to a Bézier surface.

When it comes to the newly introduced weight of a point, we usually assume it to be greater than zero. If the weight of a point is equal to zero, the point does not contribute to the shape of the surface in any way and if the weight is considered less than zero, the convex hull property does not have to hold anymore (fig.1.7), which makes both these options either superfluous or straight forward undesirable. Even so, non-positive values can be encountered in certain special (mostly theoretical) cases.

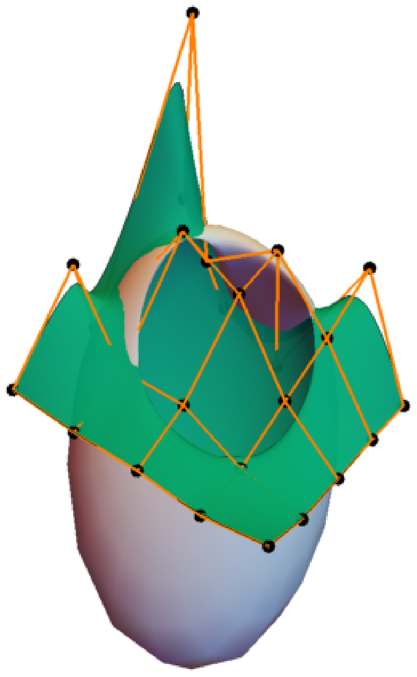


Figure 1.7: Negative weights distort the surface greatly and make it lose all its 'nice' properties.

Focusing solely on the positive values, the focal point of our interest should be what happens to the surface as the weight of a point increases. Naturally, with an increasing weight of a point the surface is drawn towards the control point until for $w_{i,j} \to \infty$ it passes directly through it while still being contained in the control net's convex hull as can be seen in the figure 1.8. Figure 1.9 shows us that we do not even need to take weights to the extreme for them to significantly influence the shape of a surface.

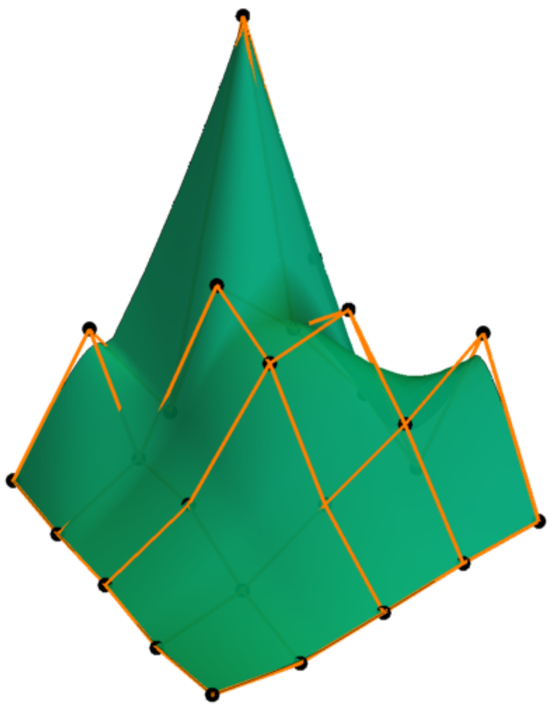


Figure 1.8: Very high weight of the highest point. The surface is drawn to this point but does not lose its important properties.

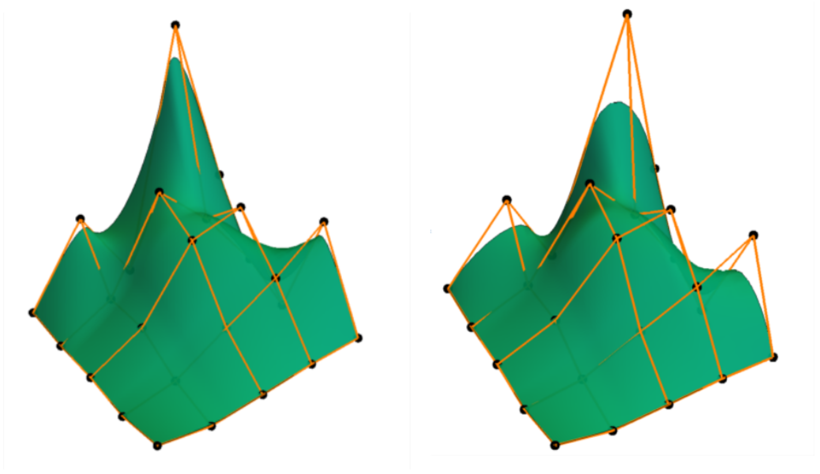


Figure 1.9: Two NURBS patches with different weights of some points.

If we choose to turn our attention to the yet unexplained part of the name of these surfaces – that being 'non uniform,' it refers to the structure of its knot vectors. When speaking of *uniform* knot vectors, we mean such vectors that their elements form an increasing arithmetical sequence, that is the difference between each two neighbouring elements of the vector is a constant. This property can but does not have to hold for NURBS. Uniform knot vectors are actually not as usual since in many cases it is desirable for the surface to pass through some of the control net's points which can be achieved via multiplicity of certain knot vector values. Furthermore, since the influence of knot vectors is significant, the non-uniformity can and often does help in achieving a greater scale of possible shapes as was shown in the section about the knot vectors of B-spline surfaces.

Before concluding this section, we show some surface patch types (other than rectangular) useful in shape modelling which can be obtained by opportunely setting their control net vertices and the knot vectors in figure 1.10.

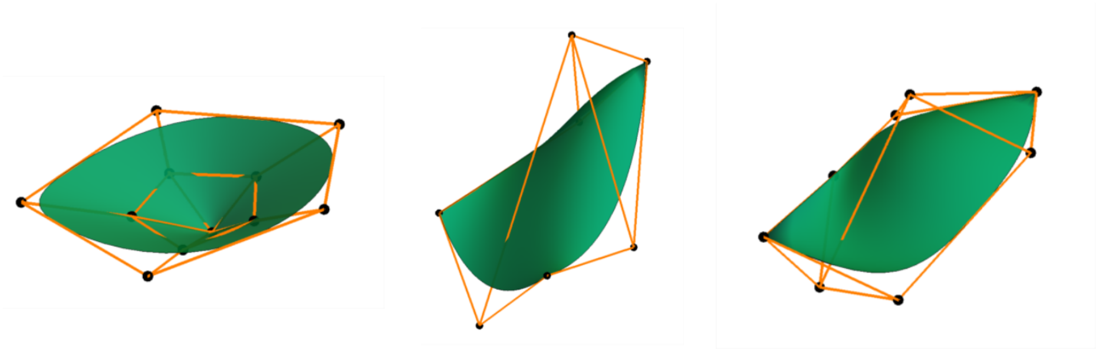


Figure 1.10: Disk patch (left), three-sided patch (middle), two-sided patch (right)

Let us now sum up some of the most important properties that made NURBS the technical standard of basically all CAD software [5]:

- The degree of a surface is at most equal to the number of control vertices in each parametric direction.
- The influence of a change of a single control net vertex is limited to $\pm k/2$, $\pm l/2$ spans in both $u-$ and $v-$direction.
- A NURBS of order k, l is always C^{k-2}, C^{l-2} continuous everywhere.
- The surface follows the shape of a control grid.
- For non-negative weights the surface is contained within the convex hull of its control grid.
- The surface is invariant not only under affine but also under projective transformations, that means that the surface can be transformed by only applying a projective transformation on its control net.
- The variability added by weighting points can be used to produce a wider scale of shapes with little to no need for plating unless the shape is very complex. (An example of a surface made of several NURBS patches can be seen in fig 1.11)
- Both Bézier and B-spline surfaces can be expressed in terms of NURBS.

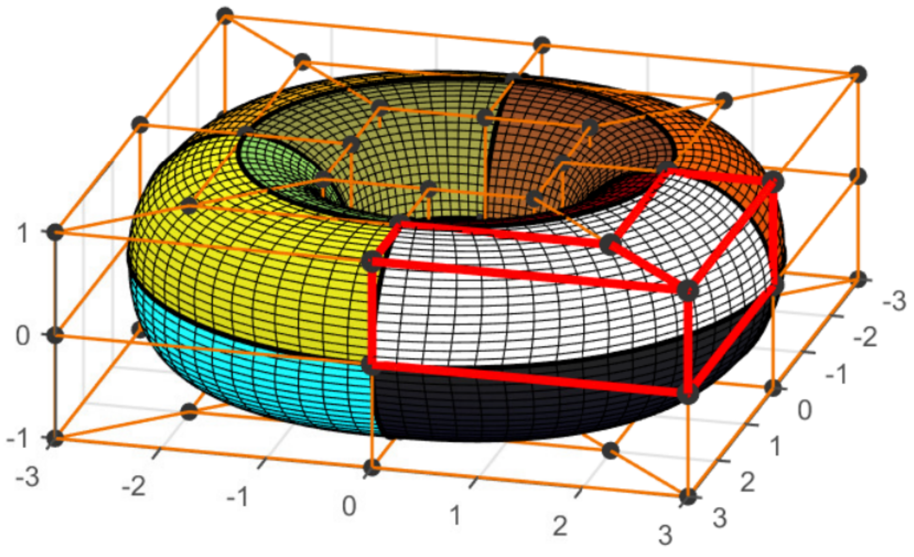


Figure 1.11: Torus made of 16 patches. (Courtesy Chudáčková [6])

1.1.4 Surfaces swept into existence

This subsection presents a strong technique of surface modelling which is often used to create 3D surfaces – it is called *sweeping*. Despite focusing solely on the NURBS here, sweeping works just as well for both B-spline and Bézier surfaces as it always defines the control net of a surface.

When we talk about sweeping we mean traversing a geometric entity, mostly a curve or a polygon, along a path (sometimes referred to as *spine curve*) in a three-dimensional space. This can be for instance a circle along a curve yielding a pipe surface (fig. 1.12).

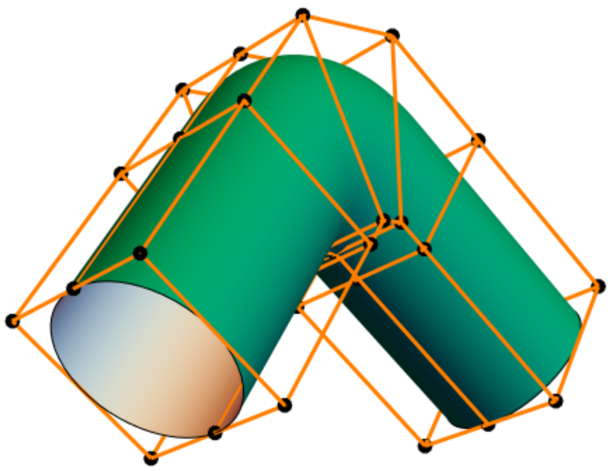


Figure 1.12: A product of sweeping can be a pipe surface.

Open curves, closed polygons or closed curves in the form of NURBS are often used to obtain sweep surfaces. Most geometric modelling and design software uses sweep surfaces enclosed by feasible end surfaces in order to create its primitives for the user to work with. Primitives like parallelepiped obtained by sweeping a square or a rectangle along a straight path, cylinders made by a circle swept along a line segment, a circle of decreasing radius swept along a straight line yielding a cone or a halfcircle rotated along a full circle in order to obtain a sphere, are then easily defined by NURBS surfaces.

Ruled surfaces

Ruled surfaces, that is surfaces which are created by a straight line moving along a path with a degree of freedom, have an irreplaceable role in a huge spectrum of industries for their good constructibility and rather simple description and can therefore be encountered almost everywhere. For these reasons, it comes as no surprise that ruled surfaces are amongst the most frequently used surfaces even when it comes to computer modelling. A good example of ruled surface is the hyperbolic paraboloid (fig.1.13).

When it comes to describing surfaces in terms of NURBS, there are two major ways, depending on some of the surface's properties. The first one is *extrusion*, that is translational sweeping of an unchanging NURBS curve $P(u)$ (of degree k described by its $n+1$ control points A_i and a knot vector $\vec{u}$) in a direction $\vec{t}$ by a distance of $|\vec{t}|$ given as a conveniently placed line segment (which is also a NURBS curve given by two control points and and the knot vector $\vec{v} = (0, 0, 1, 1)$).

Such a ruled surface $Q(u, v)$ is generally given by an equation $Q(u, v) = = P(u) + v \cdot \vec{t}$. If we apply our knowledge of NURBS however, we can see that the original curve and its version translated by $\vec{t}$ give us a set of control points $Q(u, 0) = P(u) \rightsquigarrow A_{i,1}$ and $Q(u, 1) = P(u) + \vec{t} \rightsquigarrow A_{i,2}$ which together the with knot vectors $\vec{u}$ and $\vec{v}$ defines the NURBS surface as:

$$Q(u, v) = \sum_{i=1}^{n+1} \sum_{j=1}^{2} A_{i,j} R_{i,k}(u) R_{j,2}(v). \tag{1.6}$$

By further manipulating the endcurves, more complex ruled surfaces can be obtained.

Alternatively, a ruled surface can be modelled as a linear interpolation between two known profiles (again given in the form of NURBS curves). For this to be possible however, both the curves have to be of the same degree, have the same knot vector and the same number of control vertices. There are ways of ensuring that all these conditions hold – raising the degree of the lower order curve and knot intersection ensuring the common knot vector which in consequence guarantee the same number of control points for both the curves. A more detailed description of these methods can be found for instance in [4].

For our purposes we just assume that all the conditions hold and we have two NURBS curves $P_1(u)$ and $P_2(u)$ which we are trying to blend into each other using line segments. Such a ruled surface $Q(u, v)$ is then generally given as $Q(u, v) = (1 - v)P_1(u) + vP_2(u)$.

If we denote the n control points of each curve $A_{i,1}$ and $A_{i,2}$ respectively, assume their common knot vector is $\vec{u}$ and use the vector $\vec{v} = (0, 0, 1, 1)$ for the v direction again, we again obtain the expression (1.7) as derived above.

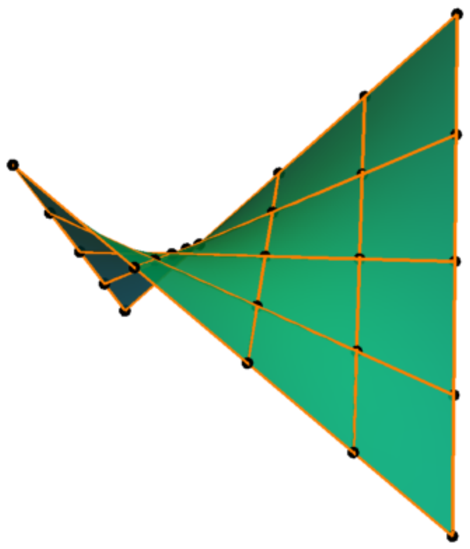


Figure 1.13: Hyperbolic paraboloid is a very common surface used in architecture.

This process can be sometimes extended and repeated for several profiles in a row. Then it is usually referred to as *lofting*. The term itself comes from the slang of shipwrights who used this method in practise when building ship hulls (1.14). Lofting itself is a very strong modelling method used by many modelling programs.

Surfaces of revolution

Another class of common and easy to describe surfaces are the surfaces of revolution. As the name suggests, these are generated by rotating a planar entity (in all generality the entity does not have to be planar but then the obtained surface can be reparametrized in such a manner that is as well could have been planar in the first place, therefore for the sake of simplification we assume it to be such from the start) about an axis in space (fig. 1.15).

If we are to obtain such a surface in the form of NURBS, we again have to start with a NURBS curve (the profile) $P(u)$ of order k in a plane containing the z-axis given by n points A_i and a knot vector $\vec{u}$.

As the first step we make six copies of each of the curve's control points in a plane perpendicular to the axis containing the chosen point, that is in such a manner that these points form a control net of a circle. Weights of individual points for a fixed i are then

$$w_{i,j} = \{w_i, w_i/2, w_i/2, w_i, w_i/2, w_i/2, w_i\}; j = 1, \ldots, 7.$$

From here the equation of a NURBS rotational surface is:

$$Q(u,v) = \sum_{i=1}^{n} \sum_{j=1}^{6} A_{i,j} R_{i,k}(u) R_{j,3}(v) \tag{1.7}$$

with knot vectors $\vec{u}$ of the profile curve and $\vec{v} = (0,0,0,\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{3}{4},1,1,1)$ of the circle [5].

Figure 1.14: Lofting is a centuries-old method for ship hull construction, now in the service of computer graphics. Source: https://www.westend61.de/en/imageView/MINF05598/man-lying-on-floor-in-a-boat-builders-workshop-working-on-a-wooden-boat-hull

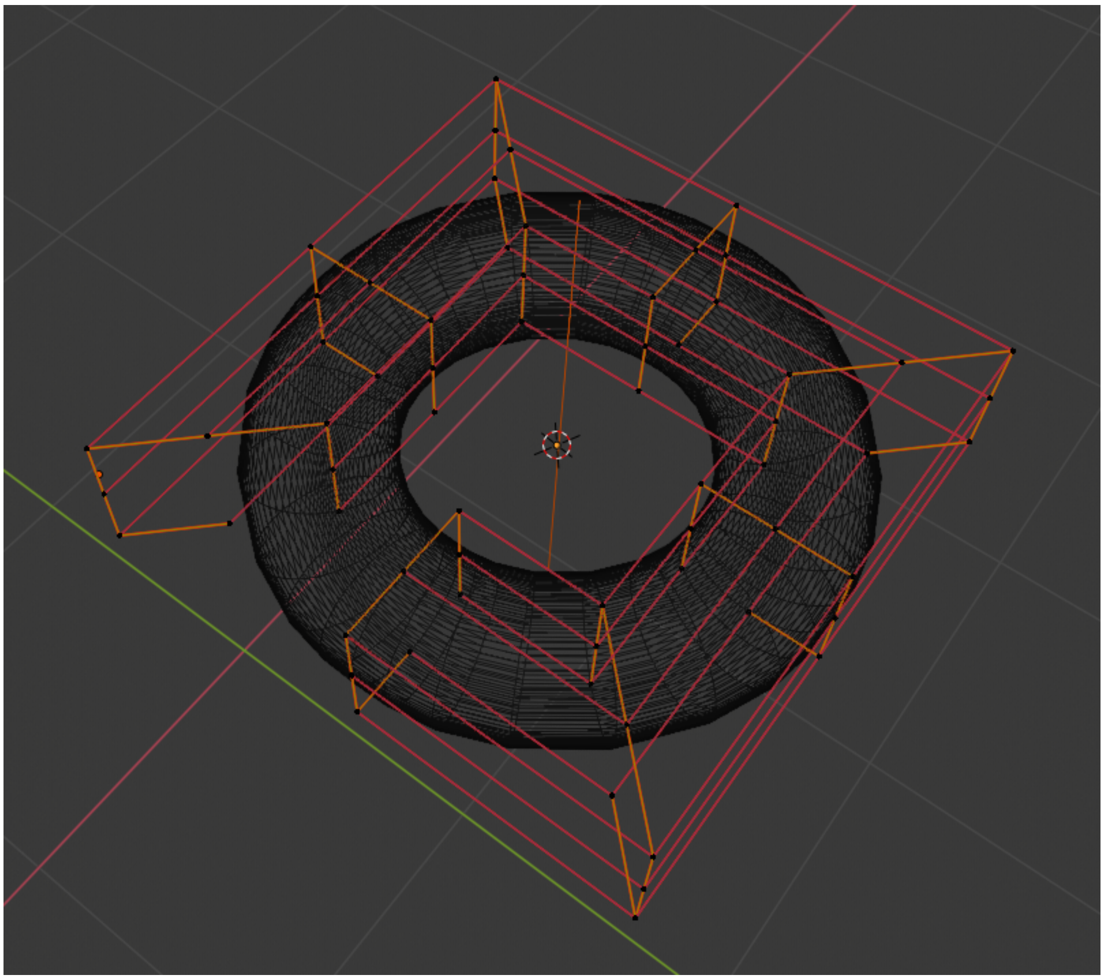
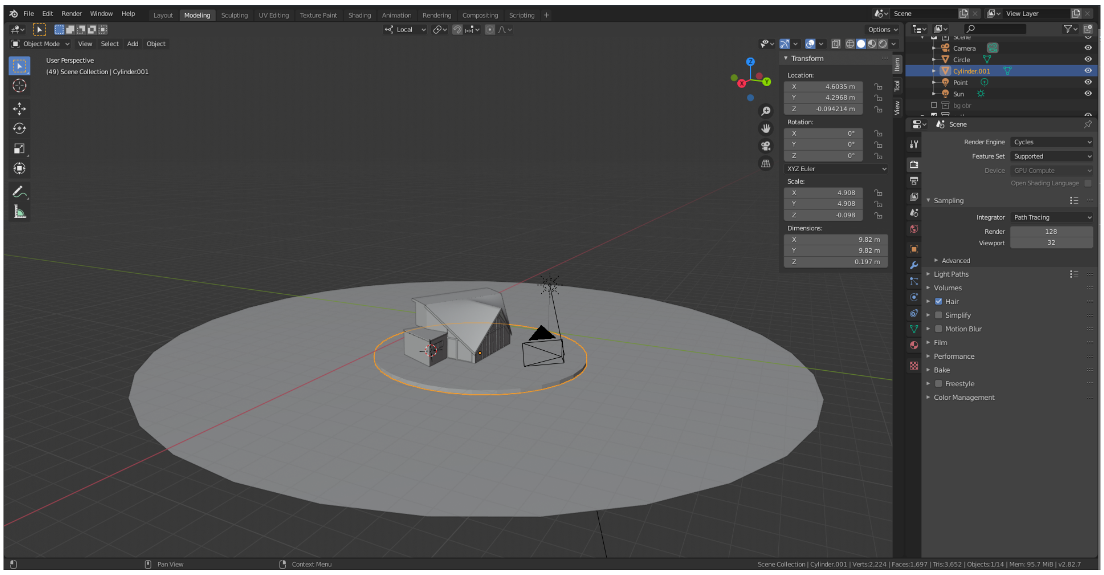


Figure 1.15: Torus is a surface of revolution with circle as its profile curve. If we take a closer look, we can see how a modelling surface tackles working with this NURBS surface, it only takes several copies of its profile curve and interpolates it with lines – it constructs is via lofting.

2. 3D computer graphics

This chapter will offer a brief look into the workings of 3D computer graphics and modelling. We will only focus on chosen areas of computer graphics (mostly those we will utilize in the last chapter) as the bare fundamentals of this topic can make for a 623-page-long book [1] or even longer [7].


File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering Compositing Scripting
Object Mode View Select Add Object
Local
Options
Scene
View Layer
Camera
Circle
Cylinder.001
Point
Sun
bg obr
Item
Tool
View
Pan View
Context Menu

User Perspective
(49) Scene Collection | Cylinder.001

Transform
Location:
X 4.6035 m
Y 4.2968 m
Z -0.094214 m
Rotation:
X 0°
Y 0°
Z 0°
XYZ Euler
Scale:
X 4.908
Y 4.908
Z -0.098
Dimensions:
X 9.82 m
Y 9.82 m
Z 0.197 m

Scene
Render Engine Cycles
Feature Set Supported
Device GPU Compute
Open Shading Language
Sampling
Integrator Path Tracing
Render 128
Viewport 32
Advanced
Light Paths
Volumes
[✓] Hair
[ ] Simplify
[ ] Motion Blur
Film
Performance
Bake
[ ] Freestyle
Color Management

Scene Collection | Cylinder.001 | Verts:2,224 | Faces:1,697 | Tris:3,652 | Objects:1/14 | Mem: 95.7 MiB | v2.82.7

Figure 2.1: Environment of Blender

The process of 3D computer graphics usually starts with obtaining a 3D model. Most common way of obtaining 3D models are through the workings of a 3D modelling program (environment of one – Blender can be seen in figure 2.1). These allow the user to visualize and interact with the model, shaping it as the will, store the models data as well as realize rendering. There are three common ways of tackling the modelling process offered by these programs: polygonal modelling (using polygonal meshes), curve modelling (using splines and Bézier curves and surfaces), digital sculpting ([8]). These methods can also be combined as shown in figure 2.2, because some parts of the model are needlessly complicated by using burbs, mostly planar parts (fig. 2.3).

Independently of the modelling method used, a model is then usually drawn using a triangular (sometimes rectangular) mesh which interpolates the surface in such a manner that it appears smooth (or as smooth as possible). This is always a trade-off between the surface's quality and the amount of data required. Most modelling programs are able to adjust the level of detail based on the current view using subdivision algorithms.

For these reasons operations leading to displaying a mesh involve a lot of problem solving regarding triangles and their mutual relationships and are an integral part of every 3D geometric software.

Another integral part of such software are operations mediating the geometric manipulation process. It turns out that, despite these can technically be hard-coded into the graphics pipeline, are best executed and algorithmized in the form of 4D matrix operations in homogeneous space. Homogeneous coordinates are

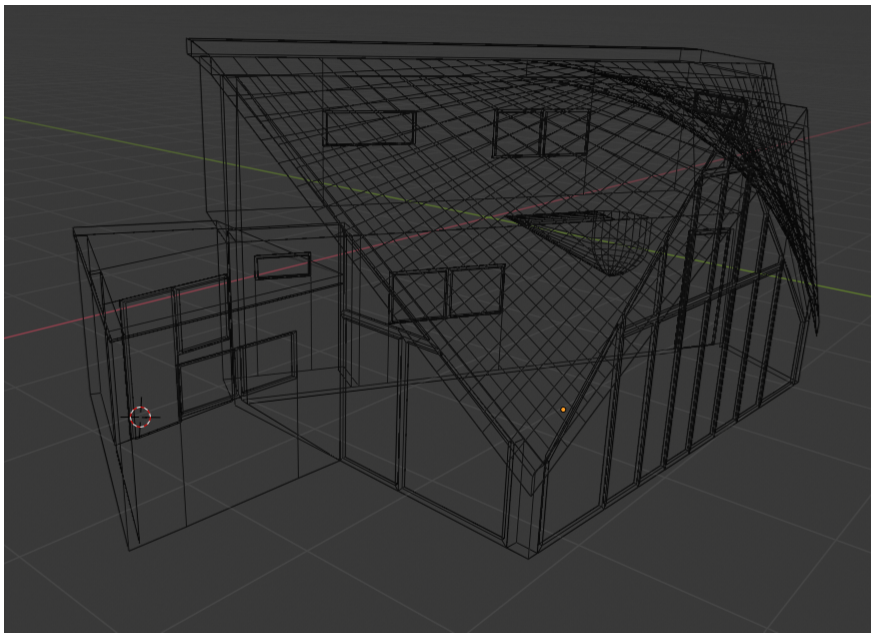


Figure 2.2: Wireframe of a house. Part of roof is made of two hyperbolic paraboloids modelled as NURBS surfaces, rest of the model is obtained with polygonal modelling.

composed of three traditional 3D coordinates (x, y, z) and a fourth homogeneous coordinate allowing us to easily handle perspective and projective viewing and transformations.

2.1 Modelling techniques in use

This section will sum up some ways and tools for obtaining a 3D model but also a part of a pipeline that leads to the model being displayed on our screen. We omit many parts of later processing such as texture mapping rendering, shading and lighting as those will not be as useful for us in later chapters.

2.1.1 Creating a 3D model

There are two main data structures used for creating 3D models: meshes and surfaces.

Meshes are collections of vertices, edges and faces which we manipulate to obtain the desired model. Surfaces were shortly introduced in the previous chapter.

Both have their pros and cons. Meshes are well suited for managing details, are faster rendered and adjusted, surfaces, such as NURBS or Bézier, outperform meshes in smoothness, preciseness and required storage data.

For their properties surface modelling is still very common in most industries. Virtual applications such as games, virtual and augmented reality on the other hand can only be computed in real time thanks to working with meshes.

The growing knowledge regarding meshes and options allowing to work with way more data slowly makes meshes more and more suitable for new enterprises.

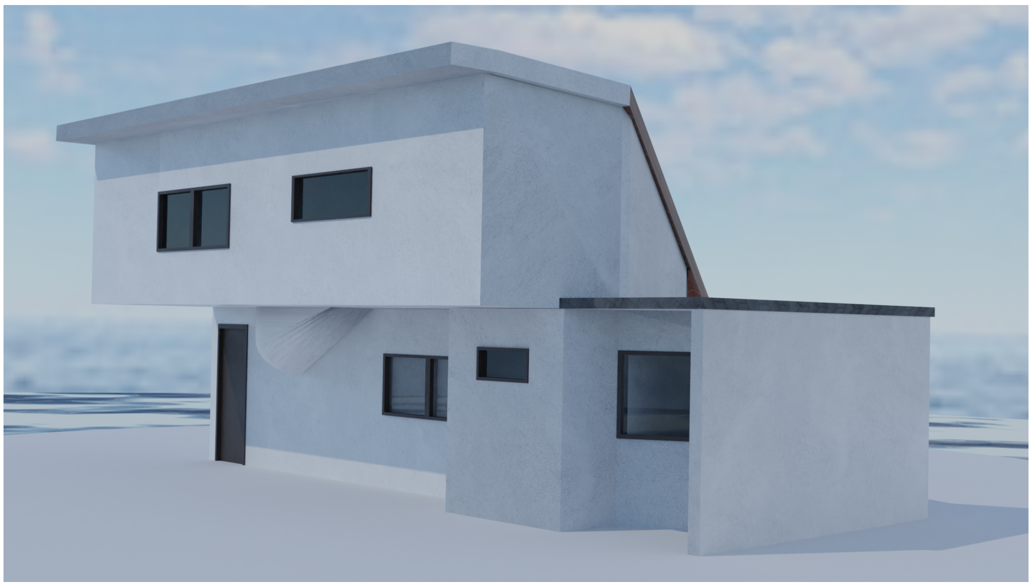


Figure 2.3: Planar faces get unnecessarily complicated if we use nurbs for modelling them. Some parts would needlessly suffer from poor quality if meshes were used for them, the overhand part 'holder' is modelled as a NURBS sphere combined with a conoid (ruled surface).

The modelling process for both however is very similar and we will sketch it up in this section with a taste of the underlying math.

2.1.2 Primitives

Geometric primitives are the simplest objects which a graphic software can handle, draw and effectively store. Even though the most basic primitives for 3D graphics are points, line segments and triangles, in practice a user of a graphic program is usually offered a way larger scale of primitives. A point is the simplest data structure a geometric software can store with no dimensions (0-dimensional). All other shapes in theory consist of an infinite number of points. Digital memory however is finite and can store only sample sets of these points. These then facilitate interpolation of the shape when it needs to be displayed or analysed.

Common primitives of 3D graphics outside of those already mentioned are polygons, cubes, spheres, cylinders, cones and tori – all these usually come in the form of meshes. More efficient on data storage and better when we aim for smoothness is a class of smooth curves and surfaces usually in the form of Bézier and NURBS. These usually come either in a general form or preprepared in form of some specific shape, common are Bézier and NURBS circles, NURBS cylinder, sphere and torus.

In Figure 2.4 we can see an example collection of spatial primitives of Blender.

Modelling process usually starts by defining the first primitive. Only via their manipulation do we achieve the shapes we wish to. This is typically done through boolean operations and 3D transformations.

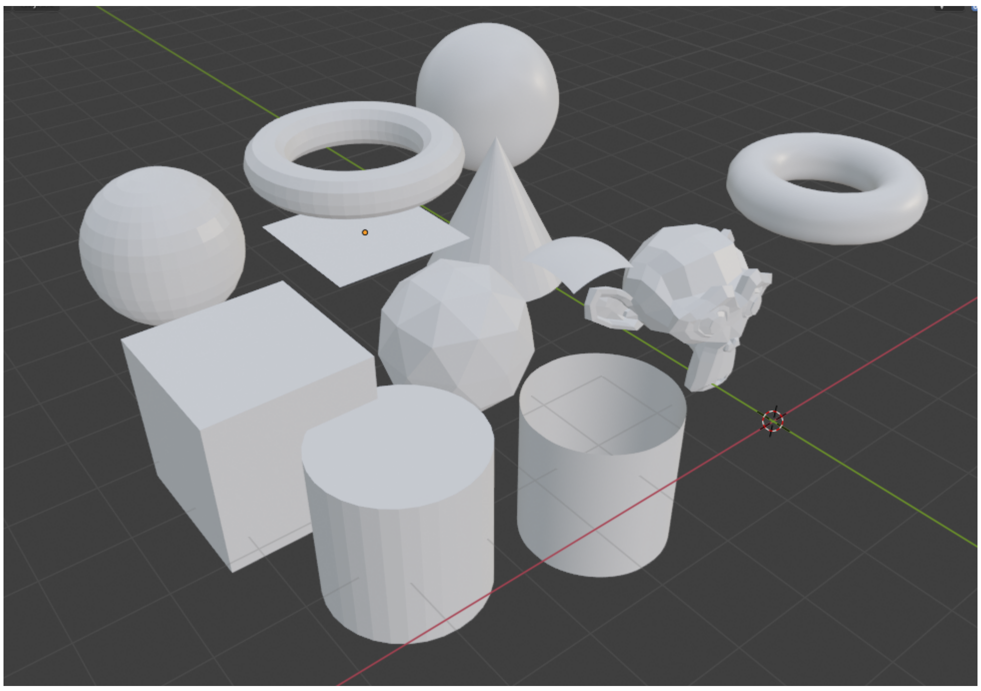


Figure 2.4: Collection of Blender primitives includes Blender's mascot: Monkey Suzanne.

2.1.3 Boolean operations

All enclosed primitives are oriented, that is normal vectors of their faces are all pointing outwards. This comes handy when we want to employ boolean operations and is yet another argument for starting the modelling process with a primitive rather than a fundamental unit such as a point. If we started with a point, extruded[1] it into an edge, then extrude the edge into a face and in similar fashion constructed a solid such as a cube, we would have to take a lot of time and effort to figure out how to execute the construction in order to make all the normal vectors point outwards (or all inwards). If we did not do that, many operations with the said object, including Boolean operations, would fail.

When we speak of Boolean data type we usually mean a data type with two possible values 1 (true) and 0 (false). Whole algebraic structures can be built upon this modest base. Important for us are the three Boolean operations, that is conjunction, disjunction and negation. In the context of 3D objects in computer geometry we usually call conjunction intersection ($\cap$), disjunction union ($\cup$) and negation ($\neg$) inversion is represented by flipping the normal vectors of a surface. A third common operation within this scheme is difference ($\setminus$) which is an intersection of one object with the inversion of the second. Generally these operation are defined:

$$A \cap B = \{x : x \in A \land x \in B\}$$

$$A \cup B = \{x : x \in A \lor x \in B\}$$

$$A \setminus B = \{x : x \in A \land x \notin B\}$$

[1]Extrusion is an operation during which an object gets copied, its vertices get connected to their corresponding images and eventual faces are formed between copied and original edges.

Boolean operations are primarily defined for solids (Constructive solid geometry – CSG). Many modelling programs however found a way around this by orienting the space with respect to an enclosed mesh or surface.

We can orient each enclosed model in such a manner that we assign 0 to the area outside of it (part of space into which the normal vectors point) and 1 to the area inside. We consider $x \in A$ if it lies in the area which got 1 assigned.

Intersection assigns 1 to the area shared by both solids. Union to the area occupied by either of these and difference $A \setminus B$ to the area occupied by A that is not in B. We only display the area which was assigned 1. We present this in the figure 2.5.

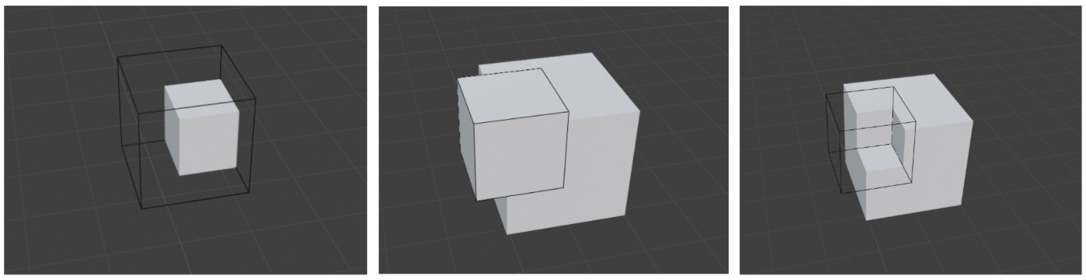


Figure 2.5: Solid part in each picture represents the result of a boolean operation between two cubes A (bigger), B (smaller), from left to right: $A \cap B$, $A \cup B$, $A \setminus B$.

Boolean operations can change the topology of an object unlike mere transformations. Change in topology can also be achieved via combining 3D transformations with operations like extrusion, deletion or addition of new points, vertices and faces.

2.1.4 3D transformations

Transformations are operations used to change the shape, position and orientation of an object. CAD programs usually do this by the means of transformation matrices.

We can change the coordinates of a point or any 3D vector representing an object in 3D space:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_{11}x + a_{12}y + a_{13}z \\ a_{21}x + a_{22}y + a_{23}z \\ a_{31}x + a_{32}y + a_{33}z \end{bmatrix} \tag{2.1}$$

Such matrices can be used to change the given vectors in various helpful ways. Every general transformation such as the one presented by 3×3 matrix in (2.1) can be decomposed into simple transforms through a process called *singular value decomposition* (SVD) which allows us to represent a matrix as a product of two orthogonal and one diagonal matrix. [1] We will therefore start with some basic spatial transformations.

Scaling

Scale is a very basic transform which changes the lengths in a given direction by a given factor (s_x in the x direction and so on) and is represented by a matrix of form:

$$\text{scale}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

If $s_x = s_y = s_z$ the scaling is said to be uniform, that is the ratios do not change. They do not need to be equal however (Figure 2.6). Scale matrices are always diagonal.

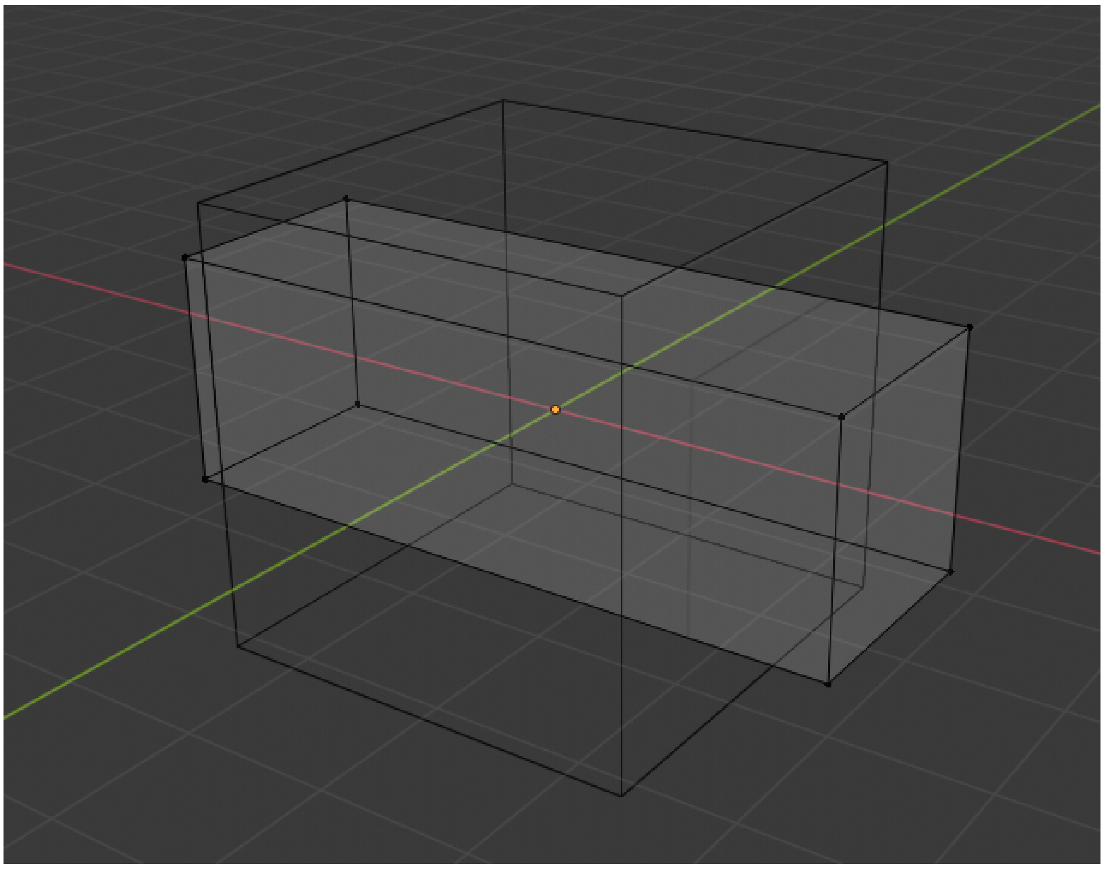


Figure 2.6: Two mutually scaled rectangular parallelepipeds: The cube is the result of scaling the other object by factors $s_y > 1$, $s_z > 1$ and $0 < s_x < 1$.

Scaling with $s_x = s_y = s_z = 1$ is called *identity* and for $s_x = s_y = s_z = -1$ it is sometimes called *point reflection*.

Shearing

Shear is a transformation that sort of pushes object sideways as can be seen in fig. 2.7.

We can perform shear along an arbitrary axis, that is if we do shear along the x axis for instance, only change the x coordinate and so on:

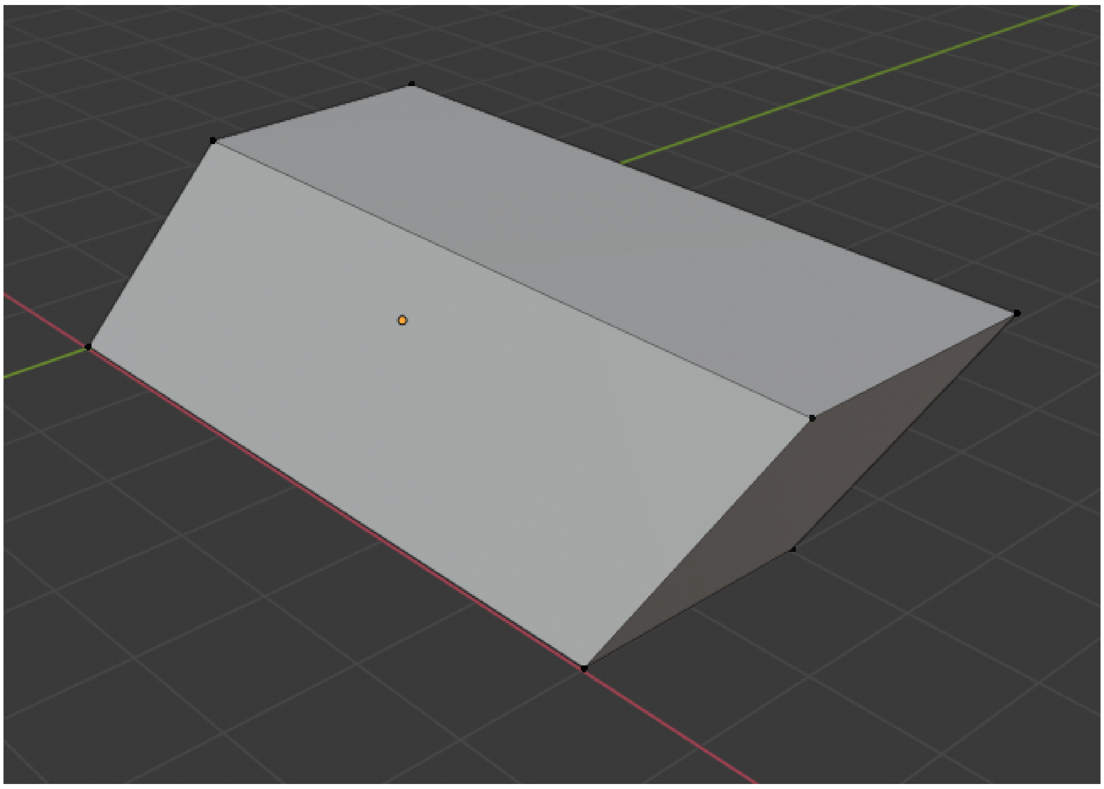


Figure 2.7: General Parallelepiped is a result of shearing. Here we see combined shear along the xy plane.

Shear along x:

$$\text{shear-x}(d_y, d_z) = \begin{bmatrix} 1 & d_y & d_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Shear along y:

$$\text{shear-y}(d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 \\ d_x & 1 & d_z \\ 0 & 0 & 1 \end{bmatrix}$$

Shear along z:

$$\text{shear-z}(d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{bmatrix}$$

Rotation

When we want to execute a rotation we first can choose between three different axes – the coordinate ones. General rotations are a bit harder and we will mention them in a section below.

Again, if we rotate about the x-axis, the x coordinate will stay the same, we basically perform a planar rotation on the remaining coordinates:

φ degree rotation about the x-axis:

$$\text{rotate-x}(\varphi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi) & \cos(\varphi) \end{bmatrix}$$

φ degree rotation about the y-axis:

$$\text{rotate-y}(\varphi) = \begin{bmatrix} \cos(\varphi) & 0 & -\sin(\varphi) \\ 0 & 1 & 0 \\ \sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix}$$

φ degree rotation about the z-axis:

$$\text{rotate-z}(\varphi) = \begin{bmatrix} \cos(\varphi) & -\sin(\varphi) & \\ \sin(\varphi) & \cos(\varphi) & \\ 0 & 0 & 1 \end{bmatrix}$$

Transformation composition and inversion

We can compose transformations in a simple manner, that is by matrix multiplication, which is yet another advantage of this representation. As matrix multiplication is not generally commutative, neither is transformation composition. Another important thing to note is in what order we multiply the transformations.

Example 3. *Consider point $p = [1, 0, 0]^T$. We want to apply two transformations, specifically two rotations: first rotation about the y-axis by $30°$ and then rotation about the x-axis by $60°$. First we apply the rotation about the y-axis. (fig. 2.8)*

$$\begin{bmatrix} \sqrt{3}/2 & 0 & -1/2 \\ 0 & 1 & 0 \\ 1/2 & 0 & \sqrt{3}/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 \\ 0 \\ 1/2 \end{bmatrix}$$

Then to this newly obtained point we apply the other rotation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/2 & -\sqrt{3}/2 \\ 0 & \sqrt{3}/2 & 1/2 \end{bmatrix} \begin{bmatrix} \sqrt{3}/2 \\ 0 \\ 1/2 \end{bmatrix} = \begin{bmatrix} \sqrt{3}/2 \\ -\sqrt{3}/4 \\ 1/4 \end{bmatrix}$$

If we write these two into one equation we get this:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/2 & -\sqrt{3}/2 \\ 0 & \sqrt{3}/2 & 1/2 \end{bmatrix} \left(\begin{bmatrix} \sqrt{3}/2 & 0 & -1/2 \\ 0 & 1 & 0 \\ 1/2 & 0 & \sqrt{3}/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) = \begin{bmatrix} \sqrt{3}/2 \\ -\sqrt{3}/2 \\ 1/4 \end{bmatrix}.$$

By matrix multiplication associativity we can get rid of the parentheses and can notice that the transformation applied first comes second in the final equation, this is vital to remember when composing transforms.

Let us also show that the order of transformations matters by switching the order of transformations:

$$\begin{bmatrix}\sqrt{3}/2 & 0 & -1/2\\ 0 & 1 & 0\\ 1/2 & 0 & \sqrt{3}/2\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 1/2 & -\sqrt{3}/2\\ 0 & \sqrt{3}/2 & 1/2\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}=\begin{bmatrix}\sqrt{3}/2\\0\\1/2\end{bmatrix}.$$

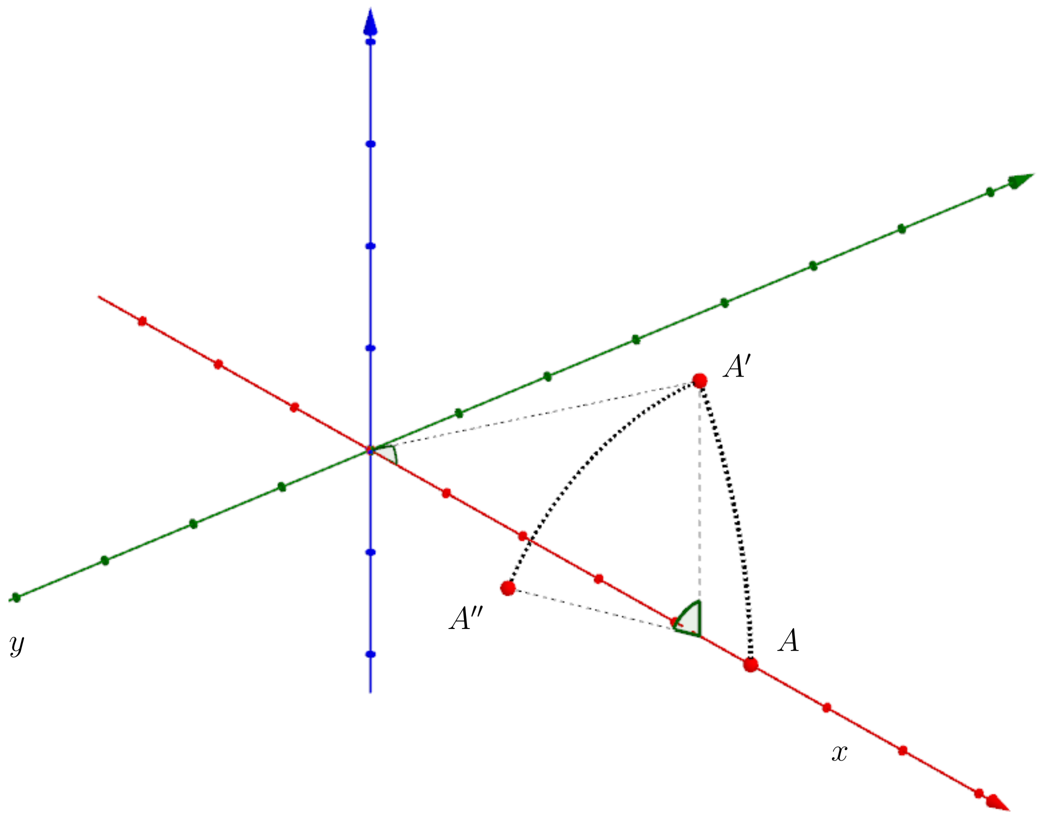


Figure 2.8: A' is the image of point A if we first apply the rotation about the x-axis, A'' if we first apply to rotation about the y-axis

When it comes to inverse transformations, it is simply expressed by an inverse matrix. Computing an inverse matrix is easily algorithmized but when it comes to understanding it, we merely need to understand what were the original transformations: inverse operation to rotation by φ is rotation by $-\varphi$ (in fact this can be always achieved by mere transposition), inverse to scaling by s_x, s_y, s_z is scaling by $1/s_x, 1/s_y, 1/s_z$ and for shear along the x-axis given by d_y and d_z it is given again by shear along the x-axis with $-d_y$ and $-d_z$.

General 3D rotation

Rotation matrices are always orthonormal, that is their three columns (and rows) are three mutually orthogonal vectors. If we denote a matrix rotating about an arbitrary vector:

$$R = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix},$$

and $u = [a, b, c]^T, v = [e, f, g]^T, w = [g, h, i]^T$, it has to hold that $u \cdot u = v \cdot v = = w \cdot w = 1$ and $u \cdot v = u \cdot w = v \cdot w = 0$. That means if we multiply this matrix by its arbitrary row, we obtain a vector of the canonical basis, for example:

$$Ru = \begin{bmatrix} u \cdot u \\ u \cdot v \\ u \cdot w \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

That means R takes the uvw basis to corresponding Cartesian axes. An inverse matrix for an orthonormal matrix always exists and is its transposed matrix: $R^{-1} = R^T$. R^{-1} represents the inverse transform to R, that also means it takes any canonical vector to u, v or w:

$$\begin{bmatrix} a & d & h \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = u.$$

From here we can deduce that if we wish to rotate about an arbitrary vector unit p, we can merely transform an orthonormal basis with $p = u$, rotate it within the frame of the canonical basis and map it back to the uvw basis:

$$\begin{bmatrix} a & d & h \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & -\sin(\varphi) \\ 0 & \sin(\varphi) & \cos(\varphi) \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

In the R matrix we only know the first row (u), the other two can be arbitrary but have to for an orthonormal basis. We can do that for instance by choosing another arbitrary vector t and then we obtain $v = \frac{u \times t}{||u \times t||}$ and $w = u \times v$ (fig. 2.9).

Rotation by the angle of 180° is sometimes called reflection through a line.

Translation – A case for the fourth dimension

The in our heads probably easiest transformation – translation behaves somewhat differently to the rest. All the transforms above could be expressed in the form

$$x' = ax + by + cz$$
$$y' = dx + ey + fz$$
$$z' = gx + hy + iz.$$

Translation by a vector $[t_x, t_y, t_z]^T$ however generally has this form:

$$x' = x + t_x$$
$$x' = y + t_y$$
$$x' = z + t_z,$$

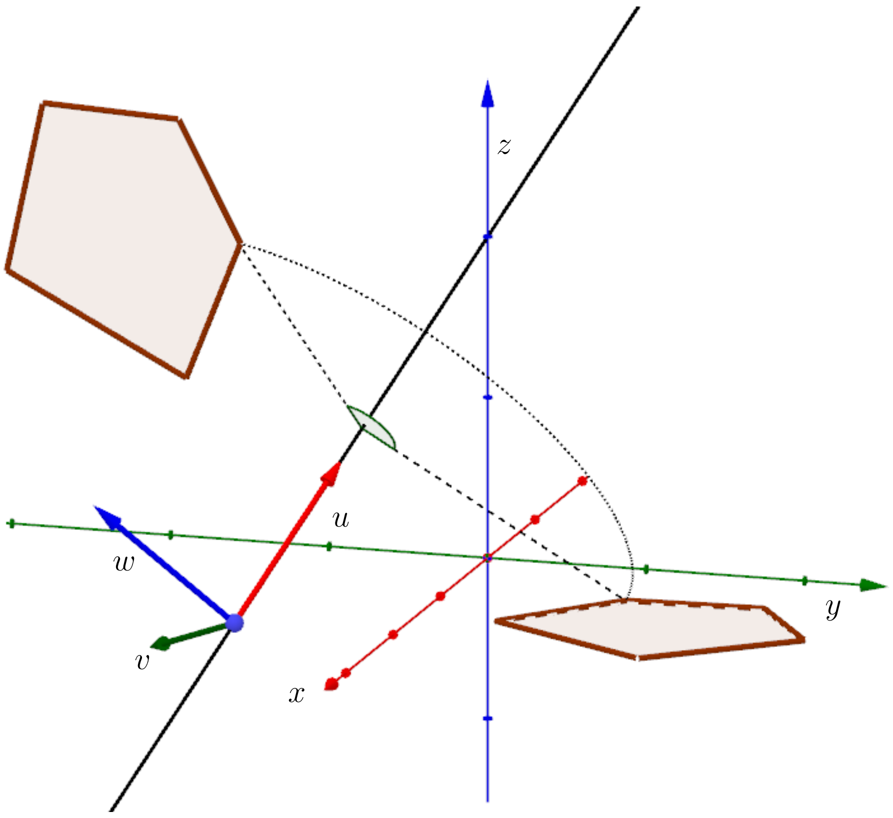


Figure 2.9: When performing a general rotation, we first transform the whole situation in such a way that the blue vector (w) aligns with the blue axis(z), the red vector (u) with red (x) and green (v) with green (y), then rotate about the x axis and transform back.

which can not be achieved by manipulating the parameters of the former. In computational geometry, when we can, we try to avoid being forced to treat one special case individually. A way to solve this issue is moving one dimension higher. If we perform a shear on the fourth coordinate we obtain:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x + t_x w \\ y + t_y w \\ z + t_z w \\ w \end{bmatrix}$$

If we now consider $w = 1$ for all points, we get what we needed.

A problem which arises with this new attitude is with vectors, we do not want these to move when we apply translation. That is solved by formally setting the fourth coordinate 0 for all vectors, the last column of the matrix will then have no influence on them. These coordinates are usually called *homogeneous*.

We do not sacrifice the functionality of the previously derived matrices by moving to homogeneous coordinates either.

Generally we can express any transformation established above as

$$\begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

there the 3×3 top left block is a matrix of composition of shearing, scaling and rotations and the first three coordinates in the last column are the translation vector. This matrix is sometimes called 3D affine transformation matrix as it encodes all possible 3D affine transformations. A general 4×4 matrix would allow for all possible projective transformations of the space.

Graphical software mostly allows us to perform these transforms with respect to two different coordinate systems. Most objects have their own coordinate system, as well as there is the global (or world) coordinate system. Even components (such as faces) of some objects usually come with their own coordinate systems which allows us to transform them independently on the rest of the said object. This allows us to apply special transformations to change the geometry of primitives. Some such transforms applied to a cube are shown in fig. 2.10.

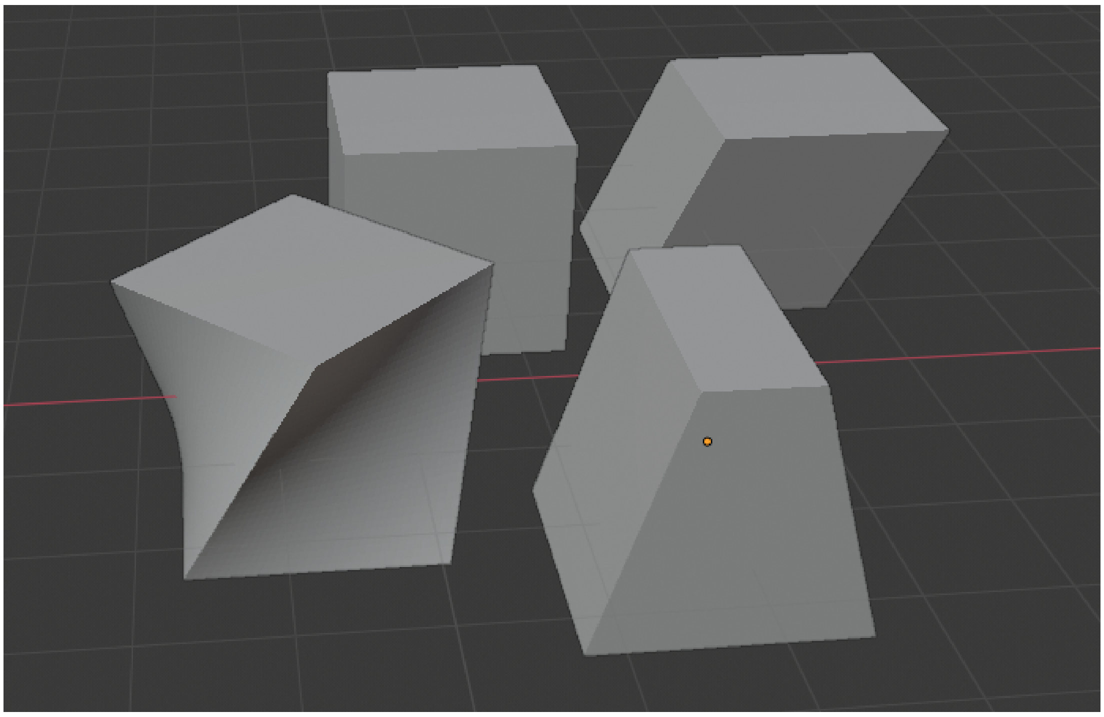


Figure 2.10: This figure shows a cube and some of its deformations obtained by manipulating its top face: twist (bottom left), taper (bottom right), shear (top right).

2.1.5 3D models on 2D screens

3D-2D projections

3D-2D projections are also handled via matrix manipulation. Here, we will finally utilize the full scope of established homogeneous coordinates when dealing with projective transformations.

When deriving the projection matrices we will often refer to certain volumes. These are manipulated and transformed in order to obtain the final projection onto a screen and represent the currently viewed scene. Anything that is not in these volumes is removed from projecting. This is done via a process called clipping, the algorithms of which can be found for instance in [9].

Canonical view volume

We will start the process by mapping the canonical view volume to the screen. The canonical view volume in a way represents the whole space that will be displayed. Starting with this step will allow us to simplify things and prepare for using the z-buffer, which helps us deal with visibility later on.

Canonical view volume is usually understood as a cube with Cartesian coordinates between -1 and 1, that is cube $(x, y, z) \in [-1; 1]^3$.

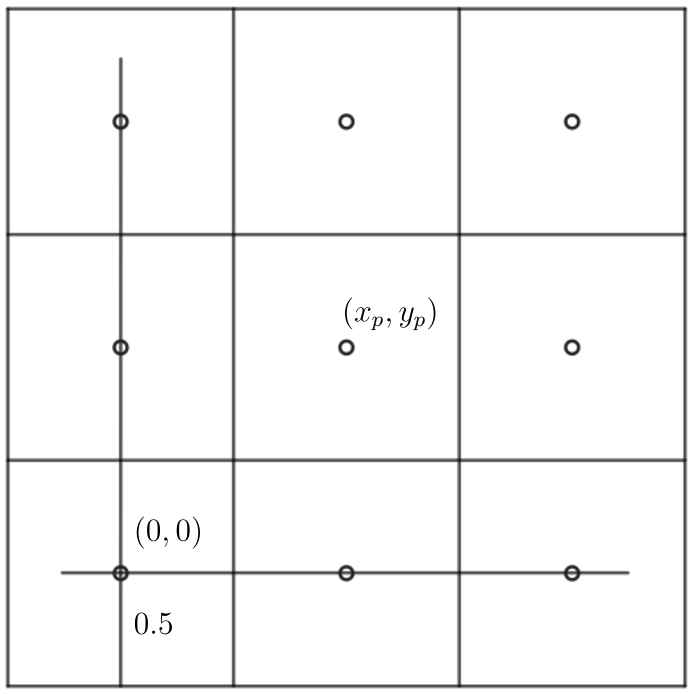


Figure 2.11: Pixels have 0.5 overshoot for which we have to adjust.

We have a screen with $n \times m$ pixels in the x and y direction respectively. It is important to note that in reality pixels are not single points, they are 1×1 squares, therefore each pixel coordinate has 0.5 overshoot for which we have to compensate, this is shown in fig. 2.11. We want to project the plane $x = 1$ to the right side of the screen, $x = -1$ to its left and $y = 1$ to the top of the screen, $y = -1$ to its bottom. This projection basically takes a square and maps it to a rectangle. We also want to adjust the coordinate system in such a manner that it aligns with the pixel coordinate system of the screen, whose boundaries are given by $[-0.5, n - 0.5] \times [-0.5, m - 0.5]$. For better orientation we will denote pixel coordinates as x_p and y_p and canonical coordinates as x_c and y_c. If we now recall what we actually did, we scaled a 1×1 square to a $\frac{n}{2} \times \frac{m}{2}$ rectangle and then moved the coordinate system to coordinates $[\frac{n-1}{2}, \frac{m-1}{2}]$ and never actually touched the z-coordinate so we will exclude it for now. In homogeneous matrix form that is:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{n-1}{2} \\ 0 & 1 & \frac{m-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{n}{2} & 0 & 0 \\ 0 & \frac{m}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

Orthographic projection

Orthographic projection is a parallel projection in a direction perpendicular to the projection plane.

Usually we would like to have a view of a volume different than canonical. The simplest case occurs when this volume is an axis aligned box $[l, r] \times [b, t] \times [n, f]$. This volume is usually called the orthographic view volume and the bounding planes are usually referred to as $x = l \equiv$ left, $x = r \equiv$ right, $y = b \equiv$ bottom, $y = t \equiv$ top, $z = n \equiv$ near and $z = f \equiv$ far. Like this we can imagine how we look at this box. The viewer stands on the xz-plane with their head pointing in the y^+ direction, looking along the the z^- direction, from which (somewhat counterintuitively) $n > f$.

We obtain the orthographic projection matrix now by merely transforming this volume to the canonical volume and then applying the viewing matrix.

Transformation of this volume to the canonical one will again have two steps: moving the centre of the orthographic volume to the centre of the canonical volume and then rescaling it to a $2 \times 2 \times 2$ cube.

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.2}$$

If we now also apply the projection matrix, we obtain the orthographic projection matrix:

$$M_O = \begin{bmatrix} \frac{n}{2} & 0 & 0 & \frac{n-1}{2} \\ 0 & \frac{m}{2} & 0 & \frac{m-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{b+t}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

By applying M_O to a point we get

$$\begin{bmatrix} x_p \\ y_p \\ z_c \\ 1 \end{bmatrix} = M_O \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \tag{2.3}$$

where the z_c canonical coordinate will be useful later for z-buffer algorithms.

Perspective projection

Perspective projection is a 3D-to-2D projection from a single point onto a viewing plane.

The same way the orthographic projection was captured by projecting a cuboid, the perspective projection is captured by an object called a rectangular frustum.

If we set the viewing plane to be the $z = n$ plane in the former notation, then the portion behind it will be scaled.

If we now manage to transform the frustum into an orthographic volume which share the points in $z = n$, we will be able to use the previously established apparatus to execute the projection again.

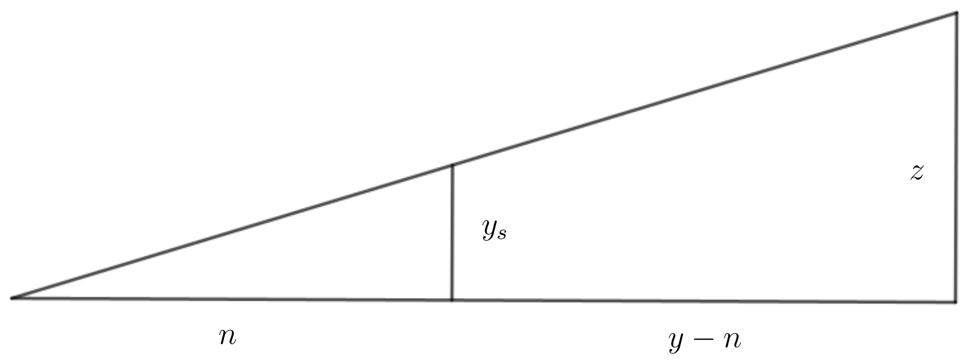


Figure 2.12: Computing y_s.

Using similar triangles as shown in fig. 2.12, we can deduce that the y-coordinate (denoted y_s) of the image of a point in the frustum can be written as (the distance of the viewer from the screen times the y-coordinate of the point being projected, that all divided by the z-coordinate of the said point)

$$y_s = \frac{ny}{z}.$$

Same thing would hold for x_s:

$$x_s = \frac{nx}{z}.$$

The z-coordinate remains unchanged.

We are looking for a 4×4 matrix that applied to a point given as $[x, y, z, 1]^T$ produces $[nx/z, ny/z, z, 1]$.

Here homogeneous coordinates are needed as the division by z would not be possible without those. Generally a point in the homogeneous coordinate system has the coordinates $[x, y, z, w]^T$ corresponding to the xyz coordinate system as $[x/w, y/w, z/w]^T$.

Example 4. *From the above follows that homogeneous coordinates differing by a* *multiplicative constant represent the same point:*

$$\begin{bmatrix}1\\2\\3\\1\end{bmatrix} = \begin{bmatrix}10\\20\\30\\10\end{bmatrix} = \begin{bmatrix}2\\4\\6\\2\end{bmatrix} \equiv \begin{bmatrix}1\\2\\3\end{bmatrix}$$

From all this we can reformulate the problem of finding projection taking $[x, y, z, 1]^T$ to $[nx/z, ny/z, z, 1]$ as $[x, y, z, 1]^T \mapsto [nx, ny, z^2, z]$. It is not hard to verify that the matrix of such projection has to have the following form:

$$M_1 = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & m_1 & m_2 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

where m_1, m_2 are yet to be determined. If we multiply the third row of the matrix by the point coordinates we get $m_1 z + m_2 = z^2$. This is a quadratic equation in z, which has at most two real solutions. We can apply constraints: it only holds true when $z = n$ or $z = f$. These are again chosen for practical reasons of simplification. If we choose these two constraints, we infer that the values of z will not change in the planes $z = n$ and $z = f$, elsewhere they change non-linearly. From here we obtain two equations:

$$m_1 n + m_2 = n^2$$
$$m_1 f + m_2 = f^2$$

from which we obtain $m_1 = f + n$ and $m_2 = -fn$.

Plugging this back to the matrix M_1 yields the projective matrix

$$M_p = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & f+n & -fn \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

This matrix can also be viewed as transformation of the frustum (projective view volume) to the orthographic view volume. (The relationship is shown in fig. 2.13) This is a big advantage as now we can simply combine it with the matrix from (2.2) to obtain what is called the projection matrix:

$$M_{proj} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix is usually a part of many application programming interfaces (APIs) though it can slightly vary depending on the orientations of axes and other variables.

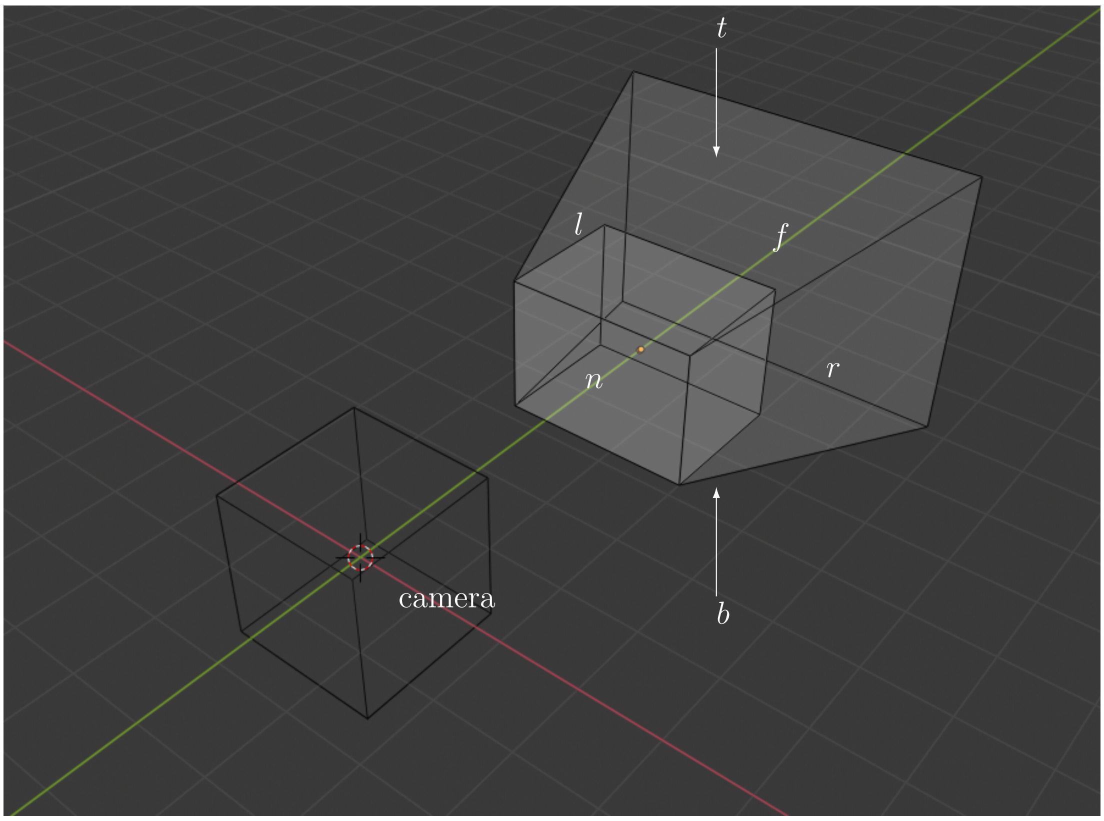


Figure 2.13: Viewing volume (wireframe), orthographic view volume (less transparent) and projective view volume (more transparent). Projection matrix takes the projective volume to the orthographic one, then to the viewing volume, where we recalculate to screen coordinates.

Camera matrix

All the previous matrices were based on view from the origin with well-aligned viewing frustums (volumes). Now if we would like to watch the scene from an arbitrary position, that is from a point e facing in the direction g with the top of our head pointing in the direction t. With these information we can easily setup a new coordinate system uvw with origin e as:

$$\begin{aligned} w &= \frac{-g}{||g||}, \\ u &= \frac{t \times w}{||t \times w||}, \\ v &= w \times u \end{aligned}$$

To be able to use the previous apparatus again we need to express a points coordinates within the uvw coordinate system and then we merely need to move the origin of the new coordinate system to $[0, 0, 0, 1]^T$ and then align axes uvw

with xyz, which yields matrix:

$$M_{cam} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Triangulations, normals and subdivision surfaces

We already mentioned that standard way of visualizing surfaces is via their triangulations. Triangulations subdivide more complex structures into a collection of simple non-overlapping triangles. This thesis focuses mainly on two types of modelling methods: polygonal modelling and parametric surface modelling.

A polygonal model gets triangulated by simply triangulating its polygonal faces, that is we only need to solve planar triangulation of polygons. When it comes to parametric surfaces, even here the triangulation is rather easy: we triangulate the planar uv coordinate system and map vertices of this triangulation onto the desired surface.

Triangulating polygons

2D polygonal triangulations are well-studied and described problems. Many algorithms varying by what we want from the triangulation. Some aim to optimize angles of the triangulation (such as Delaunay triangulation described in chapter 3), some attempt to minimize side lengths and so on. When it comes to triangulation of polygons, it is a case of a restricted triangulation, since some of its edges are already given and some can not be made. Only new edges allowed are the polygon's diagonals (fig. 2.14).

Here, we will show an algorithm by Klincsek which aims to minimize the sum of edge lengths, which uses dynamic programming.

We denote the vertices of the polygon $M_i, i \in \{1, 2, \dots, n\}$ ordered clockwise. For each diagonal from vertex M_i to M_j which lies within the polygon, we compute the minimum weight triangulation (=triangulation with minimal sum of lengths of edges) or MWT for short, of the polygon $M_i M_{i+1} \dots M_{j-1} M_j$ and store the value $L(i, j)$ of the vertex for which, while the MWT was achieved, is the third vertex of the triangle with one side $M_i M_j$, that is triangle $M_i M_{L(i,j} M_j$ (fig. 2.14). We execute this operation for i from 1 to $n - 1$ and j from $i + 1$ to n.

Once these iterations complete, we can recursively obtain the triangulation by searching through the array of stored values $L(i, j)$ initiating with $L(1, n)$. That is construct $M_1 M_n M_{L(1,n)}$, then $M_1 M_{L(1,n)} M_{L(1,L(1,n))}$ and $M_n M_{L(1,n)} M_{L(n,L(1,n))}$ and so on.

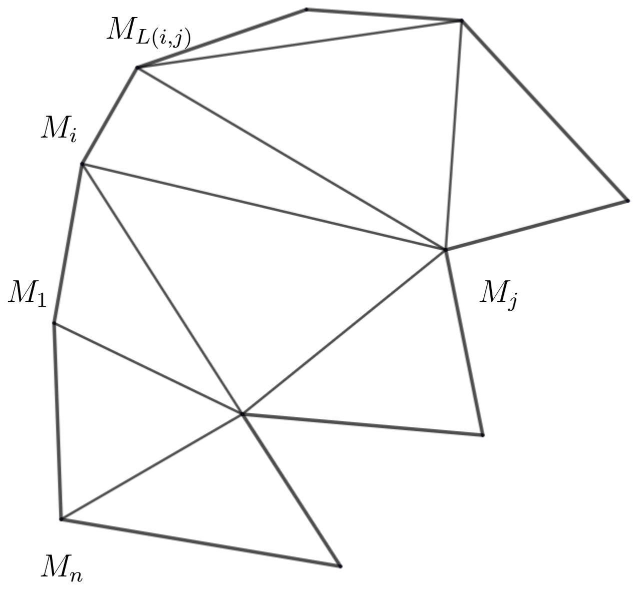


Figure 2.14: A triangulated polygon. (Not Kincsek triangulation)

Triangulating parametric surfaces

Triangulating parametric surfaces is a very straightforward process. We have already shown that a parametric surface is a projection from (u, v) to (x, y, z). When dealing with a parametric surface we simply compute some of its points with a fixed step in the u and v direction.

For $u \in [u_1, u_2]$ and $v \in [v_1, v_2]$ we choose step s in the u-direction and t in the v-direction. For this step we compute points of the surface with coordinates $[u_1 + ks, v_1 + lt]$, where $k \in \{0, \dots, \lfloor \frac{|u_1-u_2|}{s} \rfloor\}$ and $l \in \{0, \dots, \lfloor \frac{|v_1-v_2|}{t} \rfloor\}$, that is all points $p_{kl} = f(u_1 + ks, v_1 + lt)$. We will also write $m = \lfloor \frac{|u_1-u_2|}{s} \rfloor$ and $n = \lfloor \frac{|v_1-v_2|}{t} \rfloor$.

This way we rephrased our problem to the problem of triangulating an $m \times n$ net. Triangulating that is not a problem, we should only keep in mind that we ideally want all the triangles to have the same orientation that is the vertices should all be in either the clockwise or counter-clockwise order. This will help us later on when computing normal vectors.

The most optimal way to perform this triangulation is forming triangles by adding one diagonal to the non-planar rectangles $p_{ij}p_{(i+1)j}p_{(i+1)(j+1)}p_{i(j+1)}$ that are naturally produced in this process: $p_{ij}p_{(i+1)j}p_{i(j+1)}$ and $p_{(i+1)j}p_{(i+1)(j+1)}p_{i(j+1)}$ for $i \in \{0, \dots m\}, j \in \{0, \dots n\}$.

Such triangulation is sketched in fig. 2.15.

Normal vectors

Traditionally, we assign normal vectors to each triangular face. Normals are a useful tool in many processes which further manipulate with triangles, such as some algorithms for visibility, shading, reflections they also assign orientation to a model and are utilized even in further applications such as 3D printing.

Normal vectors control the orientation of a model. Ideally, we want them to face outwards from the model. This property is sometimes broken with transformations of the model. Many programs can adjust the orientation of normal

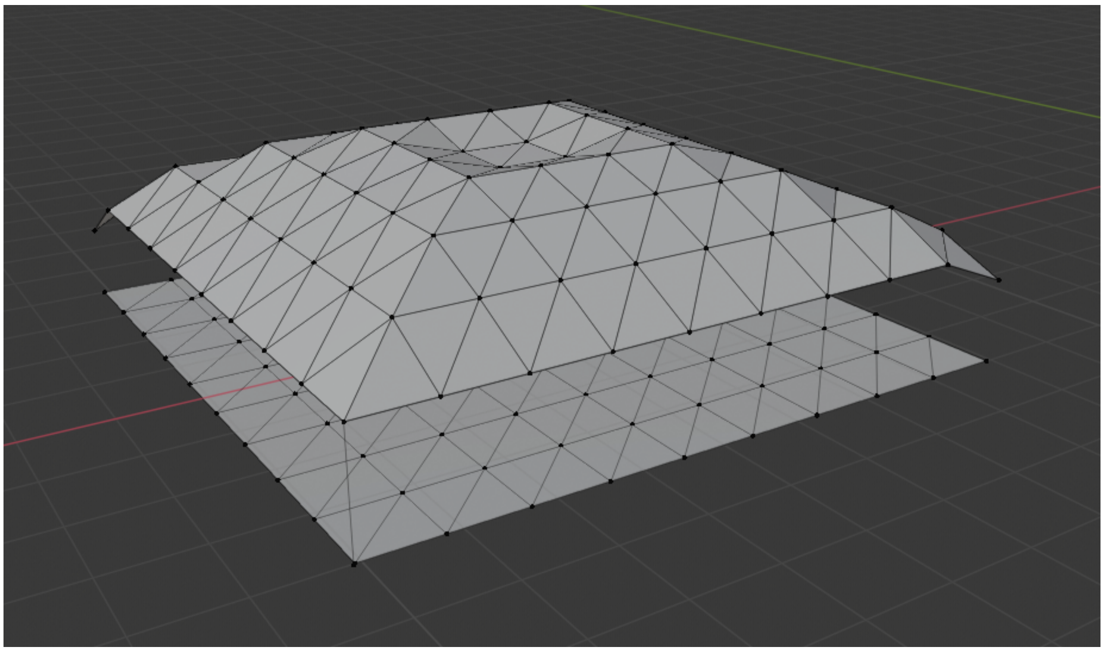


Figure 2.15: Parametric surface, not very smoothly interpolated with triangles, under it the parametric uv plane. Steps s and t are the same in both directions and fairly large.

vectors of a model if needed (either they offer tools for this or even do this automatically).

A normal vector is a unit vector perpendicular to the surface at a given point. For a general parametric surface the normal vector is given as the cross product of its first partial derivatives in the two parametric directions divided by its length: $n = \frac{\frac{\partial f(u,v)}{\partial u} \times \frac{\partial f(u,v)}{\partial v}}{||\frac{\partial f(u,v)}{\partial u} \times \frac{\partial f(u,v)}{\partial v}||}$.

In terms of triangles it is even easier to determine its normal vector. Consider we have a triangle with vertices a, b, c, two arbitrary differences of these points yield two vectors the cross product of which gives us the normal vector of the said triangle, namely:

$$n = \frac{(b-a) \times (c-a)}{||(b-a) \times (c-a)||}.$$

Normal vectors of the triangulated surface from above are shown in fig. 2.16.

Subdivision

We already mentioned that we can refine surfaces via a process called subdivision (fig. 2.17). This process does not change the topology of a surface and in the limit yields a smooth surface which would naturally consume insurmountable amount of data. Subdivision is widely utilized in the game industry and animation.

The three most common algorithms are Catmull-Clark subdivision, Doo-Sabin subdivision and Loop subdivision.

Catmull-Clarke is devised specifically for quadrangular meshes, Loop deals with triangles and Doo-Sabin is suitable for both. We will show how Doo-Sabin algorithm works:

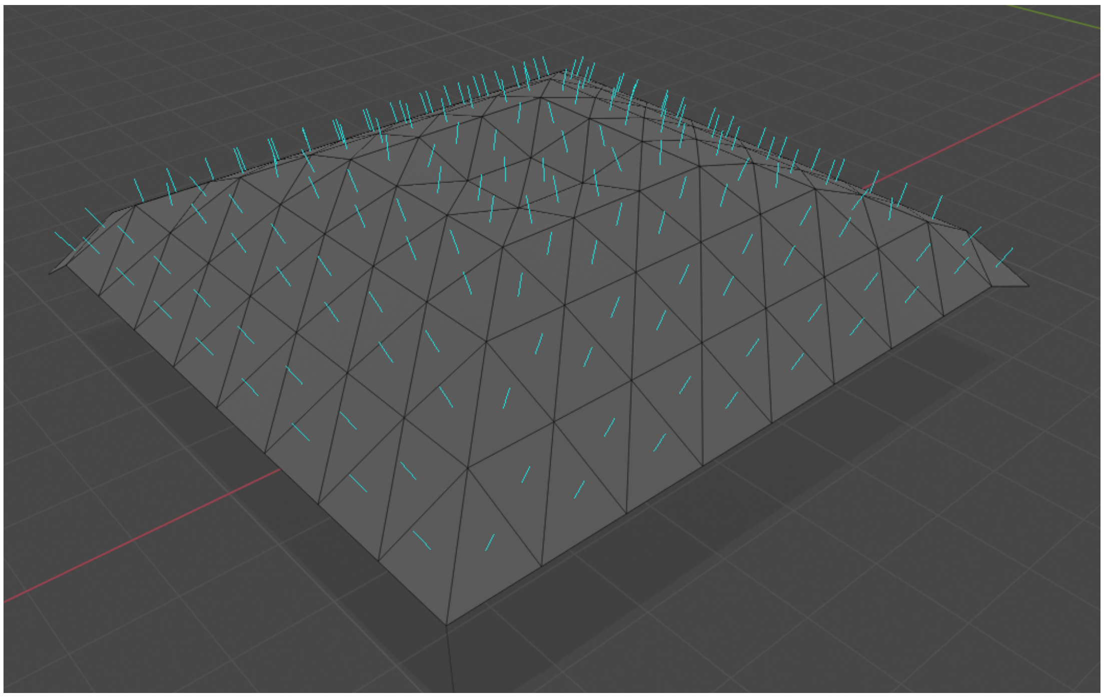


Figure 2.16: Triangulated surface with normal vectors.

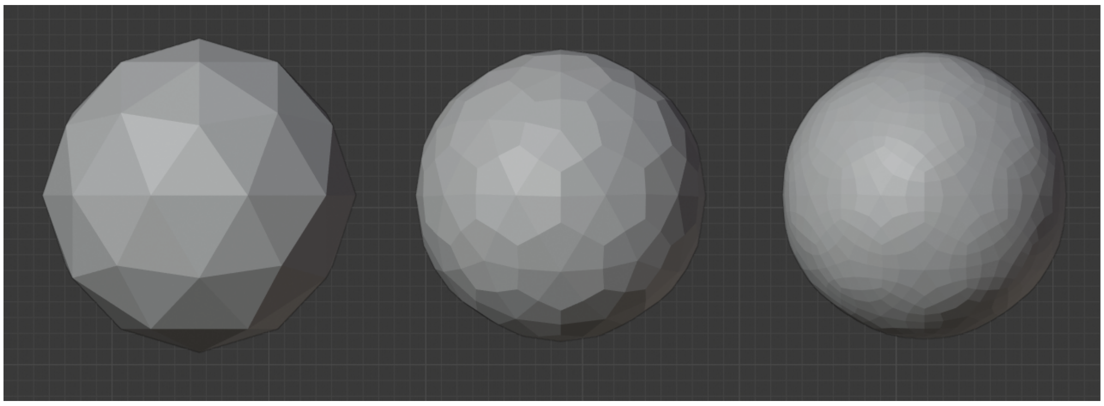


Figure 2.17: An icosphere (left), after one (center) and two (right) subdivisions

We start with an arbitrary mesh. New polygons are built by the following scheme: We identify midpoints of each edge, so called edge points. We identify all face points – centroids for each polygon of the mesh. Then a new vertex V' is formed as the average of an original vertex V, a face point of a polygon incident to the old vertex and the two edge points of this polygon's edges meeting at point V.

The new vertices are then connected – there should be two along each side of each edge by construction. Quadrilaterals are formed along each edge, n-gon is formed inside each n-gon of the original mesh and around each vertex an m-gon is obtained if m faces were incident with this point.

On this new mesh we can again iteratively apply the whole process for finer refinement.

In more mathematical terms [10], we consider point V as a vertex of the mesh with adjacent edge points $E_i, i \in \{1, \dots m\}$, $F_i^j, i \in \{1, \dots, m\}, j \in \{1, \dots m_i - 3\}$:

the face points of adjacent polygons and m_i as the number of edges of the i-th adjacent polygon, then after one subdivision we get new points $V_i', i \in \{1, \dots m\}$ as:

$$V_i' = \left(\frac{1}{2} + \frac{1}{4m_i}\right) V + \left(\frac{1}{8} + \frac{1}{4m_i}\right) E_i + \left(\frac{1}{8} + \frac{1}{4m_i}\right) E_{i+1} + \frac{1}{4m_i} \sum_{j=1}^{m_i-3} F_i^j .$$

All the mentioned subdivision algorithms work on similar principles and iteratively.

Every subdivision has to be well thought of to balance the level of detail, memory demands and computational stability. If we look from afar, the polygons do not have to be too well refined, unlike when we look up close. Most programs adjust the number of subdivisions to the current viewing position.

Visibility

Now that we know how to turn a model into a nice set of triangles, we can deal with the issue of drawing the scene realistically. The algorithms so far would simply force-draw everything in the scene, that is not what we would like. We only want to draw what we are able to see from our current camera view. We need to eliminate hidden surfaces. The predominant way of doing is *z-buffer* algorithm.

It was not that long ago when the z-buffer was not even considered an option for compute graphics even though its principle was known because it needs to potentially deal with large amounts of data. Even today it is worth to do some pre-processing before starting this process. This pre-processing is usually called *culling.*

Culling removes triangles which we do not need to even consider for final rasterization, that is triangles outside of the viewing frustum (viewing frustum culling) and faces of an enclosed mesh which are on the farther side from the camera (back-face culling).

Back-face culling can be achieved easily through dot product. We need to know the barycentre of a face, its normal and camera coordinates.

From known vertices a, b, c, the barycentre can be calculated as

$$p = \left[\frac{a_1 + b_1 + c_1}{3}, \frac{a_2 + b_2 + c_2}{3}, \frac{a_3 + b_3 + c_3}{3}\right].$$

Now the face is on the farther side of the mesh if the angle between vector $p - c$, where c is the position of the camera, and n is less than 90 deg. That corresponds to the dot product of these two vectors being greater than 0. All triangles for which $(p - c) \cdot n \geq 0$ are removed from the consideration when moving onto the z-buffer algorithm.

Z-buffer

Z- buffer algorithm is very simple and therefore very simple to implement. An issue with it is that it uses a lot of computational memory and it can only be used on opaque surfaces, it can not deal with transparent and translucent surfaces.

In the section where we were dealing with projections we came to the equation (2.3). There, we ended up with x_p and y_p – pixel coordinates on the screen and

z_c a number between -1 and 1. The plane $z = -1$ corresponds to the 'near' and $z = 1$ to the 'far plane' in the later frustums. That means for each projected point, we have its location on the screen and its depth value $z_c \in (-1, 1)$. A point colour should be drawn on its corresponding pixel if it is the closest point from all the points projected on this pixel.

We initialize algorithm by considering all pixel depth values to be equal to 1 and save the background colour for these pixels. Then for each polygon in our mesh (in an arbitrary order) we look at each pixel on which its points are projected and save the corresponding z_c value and colour if and only if the z_c is less than the value already stored for the pixel.

The z-buffer has many variations and usually differs between implementations if different programs.

This is not the end of the whole way leading to a model on a screen (fig 2.18), as we mentioned above, we omit some later parts of the rendering process. More on the whole pipeline from a practical standpoint when it comes to modelling itself can be found in [11]. A mathematical and algorithmic approach is described for instance in [1] and [12].

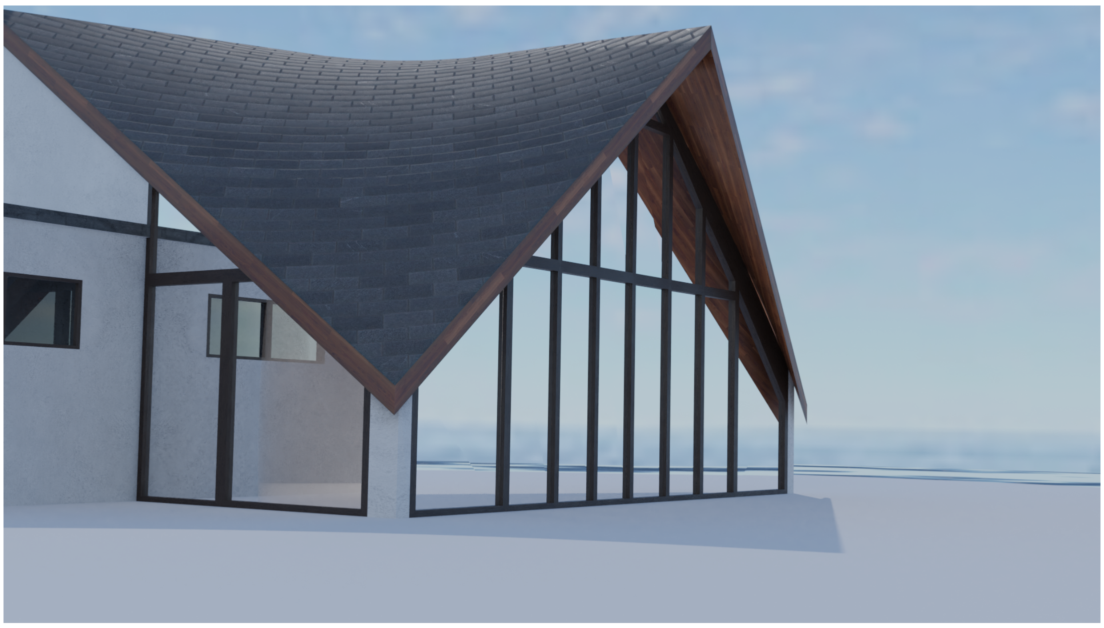


Figure 2.18: A model of a house rendered through ray tracing.

3. Photogrammetry

This chapter is dedicated to a particular method of reverse engineering called photogrammetry (fig. 3.1). Reverse engineering is a rather broad term generally understood as a process during which an object is deconstructed for the purpose of extracting some sort of desired information. In our case this will apply to recreating a real-life object as a 3D model from a set of visual and measured data.

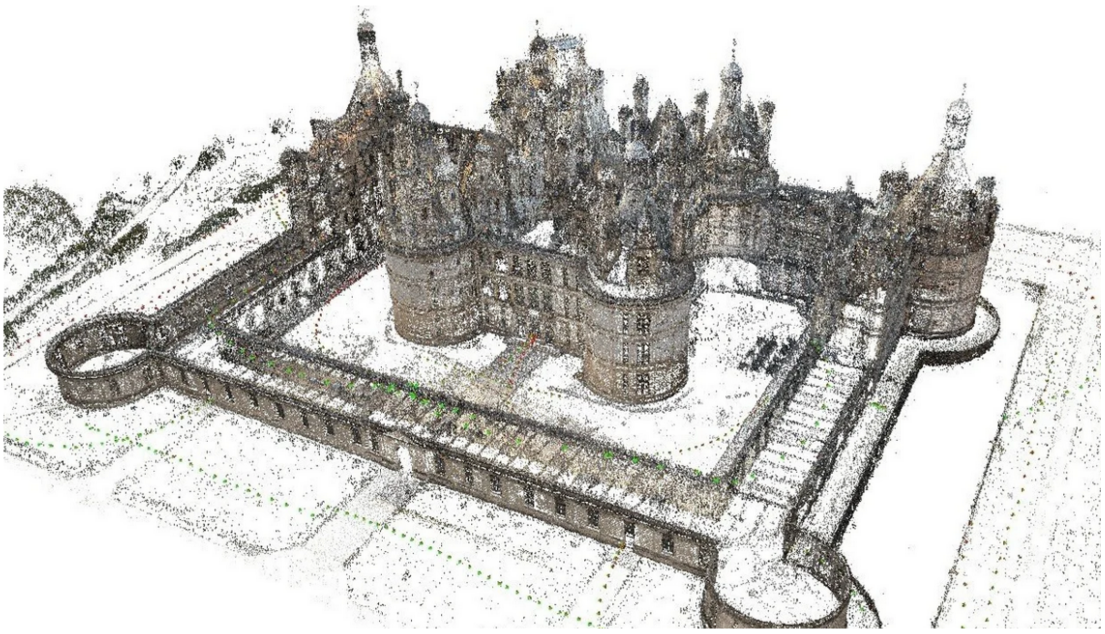


Figure 3.1: Photogrammetry can be used for 3D reconstructions of large structures. Source: `https://all3dp.com/1/best-photogrammetry-software/`

This branch of reverse engineering in itself is very broad and methods of obtaining the initial data includes methods such as 3D laser scanning, mechanical coordinate measuring and the one which we will focus on here – photogrammetry.

Despite its initial position of a relatively unreliable method compared to other, it found its way to the spotlight of many researches. This is partially due to several relatively recent innovative algorithms and partially due to a wide spread of high performing computational devices and quality cameras (mostly in the form of mobile phones) which also led to an accumulation of big datasets of images easily accessible on the Internet.

Photogrammetry in general is a science of obtaining reliable information about a physical object from photographs. For the purposes of this thesis we will consider *stereophotogrammetry* or 3D reconstruction from multiple images.

It is a *passive* method of retrieving 3D unlike 3D scanning. That means the bare images are what is needed for the reconstruction, we do not require any external intervention such as projecting patterns or other forms of light on the object of interest. Currently, the most prominent method of photogrammetry is structure-from-motion method (SfM) which will be the main focus of this chapter.

3.1 Before SfM – Finding the perfect match

The line between processes, helping to recreate a 3D situation from a set of images, which do or do not belong in the SfM pipeline is blurry and varies slightly among different authors. Here, we understand initial feature extraction and initial image matching as a (although necessary) pre-stage of the SfM process.

A short trip through digital image processing

In this short section we will briefly sum up some important knowledge needed for further reading. That is how computers represent, process and alter images.

A digital image consists of individual pixels[1], these are used as a coordinate system for orientation in the image. The coordinate system starts at the top left corner pixel, which is usually assigned coordinates $(0, 0)$ (some programs such as *Matlab* start their coordinate systems with $(1, 1)$). Considering we are dealing with an image of the size $m \times n$ pixels, the most bottom right pixel then naturally has coordinates $(m-1, n-1)$.

Thanks to this system, a computer can store each image in a form of a matrix, where each pixel with coordinates (x, y) gets assigned an integer value $f(x, y)$ called intensity. This value stores information about the pixel. In case we know we are dealing with a black-and-white (or binary) image, the convention is to encode a black pixel as 1, a white one as 0. Colour images are encoded by three matrices where each specifies the amount of red, green and blue (RGB colour system) that makes up the pixels colour. Grayscale images are usually encoded by integers between 0 and 255 for different levels of gray.

This allows a computer to operate with an image as it would with a matrix and through these operations alter it. Very basic operations can be used for altering an image. Matrix transposition will result in ninety degree rotation, adding constants to every number of the matrix leads to global colour change, switching the positions of two matrices in the RGB setup results in colour swaps and so on. The most common way of processing digital images however is utilizing kernels via a process called convolution.

For short, kernel is a matrix (usually small) which is used for blurring, sharpening and other operations on images.

Convolution is a broader mathematical term for an operation denoted usually by $*$. It can be specifically defined for matrices, which is what we need for further work.

Definition 4 (Matrix convolution). *For discrete functions $f(x, y)$ and $g(x, y)$ represented respectively by matrices $A_{m\times n}$ and $B_{k\times l}$, $f * g$ is the convolution of f and g and is defined as:*

$$f * g = \sum_i \sum_j a_{i,j} b_{x-i+1,y-j+1},$$

where $a_{i,j}$ are elements of A, $b_{i,j}$ elements of B and i and j range over all legal subscripts of $a_{i,j}$ and $b_{i,j}$.

[1]the smallest graphical element, which can have only one colour at a time

In fig. 3.2 the process of convolution is shown.

In our case $f * g$ would be the filtered image matrix and A and B would be the kernel matrix and the image matrix (it does not matter which one is which as convolution is commutative).

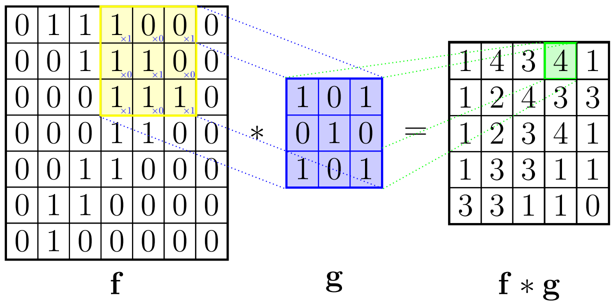


Figure 3.2: Matrix convolution example

3.1.1 Scale Invariant Feature Transform (SIFT)

When given a set of images for a 3D virtual reconstruction, we first need to know how and even whether these images relate. This is done in two steps:

1. Extracting features of interest in each image.

2. Finding and matching these features among different images.

This is mostly done through the workings of scale invariant feature transform algorithm, which meant a huge breakthrough in image feature recognition when it was introduced in 2004 by D. G. Lowe ([13]). Since then new useful methods such as Speeded Up Robust Features (SURF) were introduced ([14]), yet SIFT remains the most prevalent one.

The key idea behind SIFT is identifying blobs[2] which are invariant under transformations such as rotation, translation and scaling, and partially to other changes such as change in illumination, which occur when the viewpoints change during image acquisition. Detailed account of this process is best described in the original work [13].

In the first step the SIFT algorithm identifies scale-space extrema. Interest points are described in terms of their location (given by x and y coordinates in the image) and scale (denoted as σ).

Locations invariant under changes in scale change are then for various reasons (described in [13]) detected via applying a Gaussian scale-space kernel:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}.$$

.

We then define the scale-space as

$$L(x, y, \sigma) = G(x, y, \sigma) * f(x, y),$$

[2]regions with pixels similar to each other while distinct from their surroundings

where f is the intensity function of an image described above. For reasons based on the relationship between the scale-normalized Laplacian of Gaussian ($\sigma^2\nabla^2 G$) and the difference of scale space functions separated by a constant multiplicative factor k, Lowe suggests using scale-space extrema of said difference ($D(x, y, \sigma)$) in order to find stable key point locations. That is using

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma).$$

This method is referred to as difference-of-Gaussian (DoG), as it can be rewritten in the form of Gaussian difference convolved with the image (fig. 3.3).

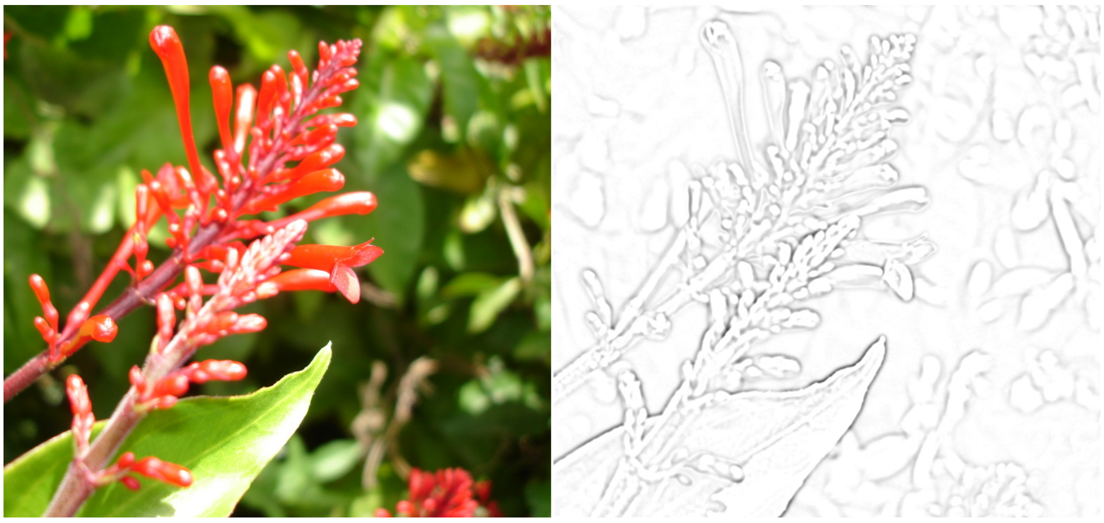


Figure 3.3: Flowers before difference of Gaussians and flowers after difference of Gaussians grayscale. Source: `https://en.wikipedia.org/wiki/Difference_of_Gaussians`

Since we know that two images (with Gaussian kernel applied to them) can be seen as matrices, the idea of image subtraction should not catch us off-guard.

An approach by which all this is realized in practice is shown in figure 3.4. The initial image is incrementally convolved with Gaussians to produce spectrums of images separated by a constant k in the scale-space. The number of steps depends on the value of k, we aim to reach 2σ with the last step. Adjacent image scales are then subtracted to obtain the DoG images.

Once we complete this for the original image, the method is repeated for a subsample of the image by considering only every other line and row of its matrix.

Detection of the extrema here is then done through going through individual points in $D(x, y, \sigma)$ which are then compared to all its neighbours in the current image (there is 8 of them) and to all the nine neighbour points that are in the scale directly above and below it (there is $2 \cdot 9$ of those: fig. 3.5). A point is considered an extremum if its value is larger or lower than the value of all the other points we compared it to.

Points identified in this procedure are only candidates for key points. Some will further be rejected – namely low contrast points and poorly localized candidates along edges.

The elimination of low contrast points is done using a Taylor expansion of

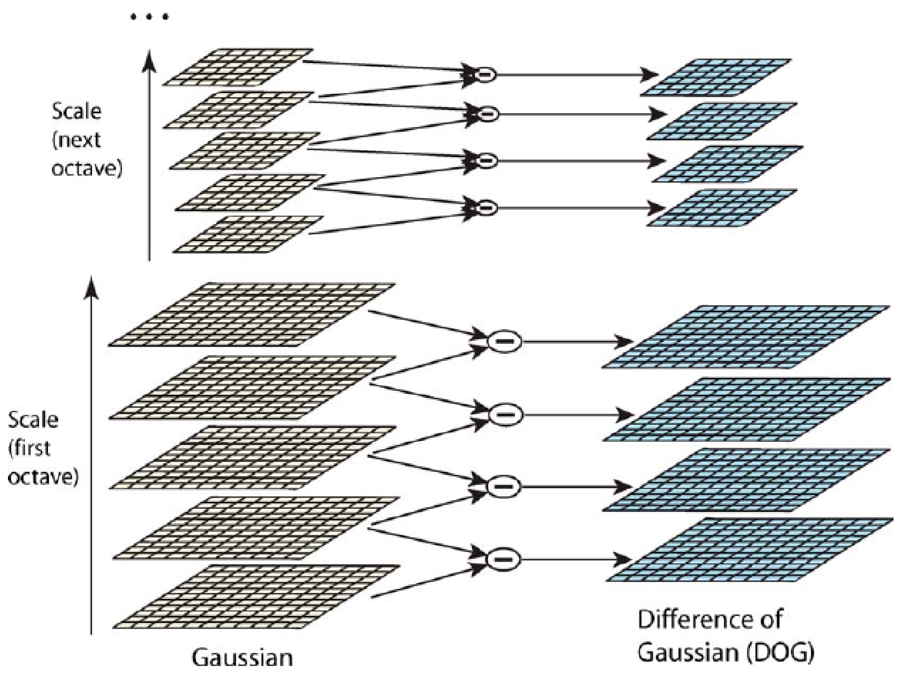


Figure 3.4: Downsizing scales of Gaussians and their differences. Source: [13]

$D(x, y, \sigma)$ up to quadratic terms:

$$D(\boldsymbol{x}) = D + \frac{\partial D^T}{\partial \boldsymbol{x}}\boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^T \frac{\partial^2 D}{\partial \boldsymbol{x}^2}\boldsymbol{x},$$

where the derivatives are taken at the sample point and $\boldsymbol{x} = (x, y, \sigma)^T$ is the offset from this point.

Some of the candidate points are then rejected if this function evaluated at its extremum is less than a certain threshold value.

The DoG function shows strong responses along edges. These are often very unstable to noise. They are rejected based on the function's Hessian. That is a 2×2 matrix H consisting of second partial derivatives at the location and scale of key points:

$$H = \begin{bmatrix} \frac{\partial^2 D}{\partial x^2} & \frac{\partial^2 D}{\partial x \partial y} \\ \frac{\partial^2 D}{\partial x \partial y} & \frac{\partial^2 D}{\partial y^2} \end{bmatrix}.$$

A candidate point is then discarded if

$$\frac{(D_{xx} + D_{yy})^2}{D_{xx}D_{yy} - D_{xy}^2} < \frac{(r+1)^2}{r}$$

for a given threshold r, where $D_{xx} = \frac{\partial^2 D}{\partial x^2}$ and so on.

To achieve rotation invariance, the remaining points, which are now the desired key points, are assigned orientation.

That is done by computing gradient magnitude $m(x, y)$ and the direction $\theta(x, y)$ of $L(x, y)$ (smooth image sample) using pixel differences ([13]):

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

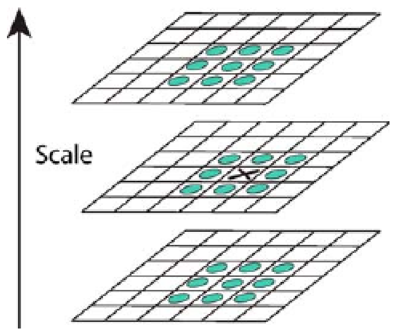


Figure 3.5: We check the point of interest against its neighbours in its layer and in two neighbouring layers. Source: [13]

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x+1,y)-L(x-1,y)}{L(x,y+1)-L(x,y-1)}\right).$$

This is computed for all points within a region around a key point and an orientation histogram of 36 bins is formed. Each sample of the histogram is weighted by its gradient magnitude and spacial Gaussian kernel with $\sigma =$ $= 1.5 \cdot \langle\text{scale of key point}\rangle$. The peak in orientation within this histogram is then assigned to the key point (more can be assigned if they are within 80 % length of the highest peak).

All this led to finding key points and assigning them scale and orientation. These parameters can be seen as a local 2D coordinate system providing a local region with an invariance to these parameters (fig. 3.6).

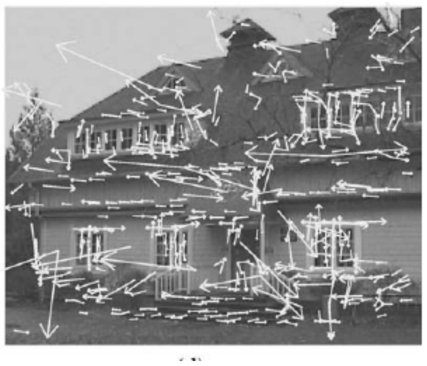


Figure 3.6: Image key point descriptor. Source: [13]

The next step is assigning local image descriptors which ensure best possible invariance to change in illumination and 3D viewpoint (that is affine or 3D projection).

This is done by using gradient orientation histograms in a 16×16 neighbourhood of a key point and forming weighted 8 bin histograms of 4×4 regions of this neighbourhood. This yields 16 8 bin (8-dimensional) histograms (fig. 3.7), the values of these concatenated lead to a SIFT descriptor of a region – a 128-dimensional vector, which is then normalized. The reasons for using this as a descriptor are specifically described in the original work ([13]) and are mostly based on experimentation with and analysis of influence of various approaches and numbers.

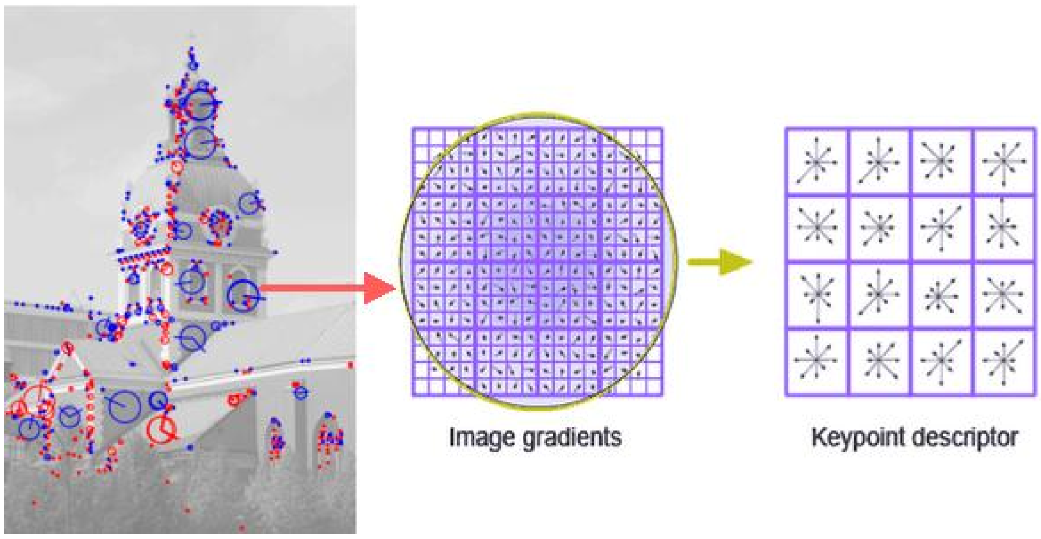


Figure 3.7: Image key point descriptor. Source: `https://www.codeproject.com/Articles/619039/Bag-of-Features-Descriptor-on-SIFT-Features-with-O`

A computer remembers all the data obtained above in a form of a SIFT feature, that is a vector computed at local extrema in the scale space $\langle \mathbf{p}, \mathbf{s}, \mathbf{r}, \mathbf{f} \rangle$, where $\mathbf{p}$ is the location of a point, $\mathbf{s}$ is its scale, $\mathbf{r}$ orientation and $\mathbf{f}$ its SIFT descriptor. The former three are view-independent (that is why we constructed them this way after all) and the latter is largely view independent.

Matching images

Having found all the blobs with distinctive features in each image of a dataset, we can now proceed and match pairs of these images (fig. 3.8).

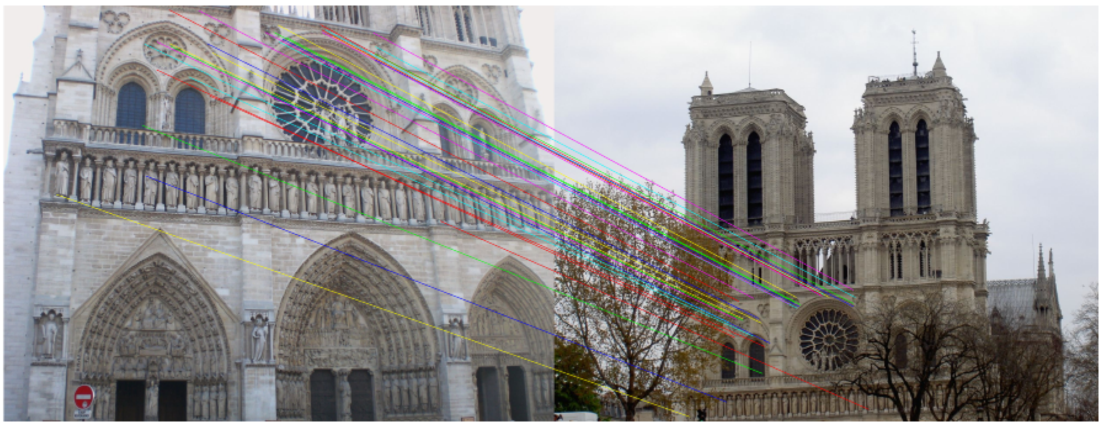


Figure 3.8: Matched features between two different views. Source: [15]

Given two images, one with n key points, other with m key points assigned, how do we match corresponding features?

This is done based on the Euclidean distance[3] between the descriptors of individual key points in each image. The points that are closer in the distance than a certain threshold are marked as a match.

This could lead to false matches however, since many parts of an image can be structurally similar. What if the second best match is actually right, or the third? If for a point in one image, the two closest points from the other image, have their distance ratio $\left(\frac{\text{distance from closest}}{\text{distance from second closest}}\right)$ greater than a given threshold, all these matches are discarded in order not to make an unnecessary mistake. The experimentally obtained threshold integrated in most systems is 0.8.

Finding the 2 closest descriptors in the second image for each feature is computationally expensive if a brute force approach is taken. Therefore many optimized algorithms exist. The most common one is Approximate Nearest Neighbour ([13]).

Even with these precautions taken, especially with repetitive patterns in the images, the algorithm can match wrong points, which could ultimately ruin the whole reconstruction.

The answer to this mostly comes in the form of statistical methods, often that is RANdom SAmple Consensus (RANSAC). This algorithm is beyond the scope of this work. In short, it utilizes a try-and-error approach. It works in three basic steps:

1. Taking a sample of matched data points required to fit a model.

2. Computing model parameters using the sampled data points. This is mostly based on the 8 point algorithm as described for instance in [16].

3. Assigning a score based on how well other points fit the model within given threshold.

These steps are repeated until the best model is found with high confidence.

Points, which do not fit this model are then removed.

Usually, if RANSAC leaves us with less than twenty matched pairs between the two images, all matches are discarded as such amount of feature correspondences is too low for us to reliably match the two images.

An example implementation made by D.Lowe himself of the whole SIFT pipeline presented, can be found on `https://www.cs.ubc.ca/~lowe/keypoints/`.

3.2 SfM

The outcome of the previous section are pairs of overlapping images and their feature correspondences.

It is worth mentioning that comparing all pairs of images of large datasets for feature matching can be computationally costly. For that reason some photogrammetry programs such as Meshroom add a step of image matching based on scalable recognition with a vocabulary tree as described in [17] which sorts and stores image feature in a tree structure. Extracted features of an image are then compared only inside a specific part of the tree and not amongst all images. When a reconstruction is done from big

[3]Euclidean distance is usually given as $\sqrt{\sum_{i=1}^{n}(f1_i - f2_i)^2}$, where $f1_i$ is the i-th coordinate of the first vector, $f2_i$ of the second and n is their dimension

samples of images, this is the only way to be able to compute everything within a reasonable amount of time.

In order to keep this section from spanning a whole thesis, we will only sketch out the whole process and leave out most of the miscellaneous algebraic steps to which we will only leave a reference for further reading.

A camera

There are several mathematical models of cameras, the most common and the one we will cover here is the pinhole camera model. From the other models, let us mention the fish eye camera model which creates similar distortions to those of a door peephole.

The pinhole camera model (fig. 3.9) describes the relationship between coordinates of a point in the 3D space and the coordinates of its perspective projection from the center of projection (pinhole) onto an image plane (virtual plane).

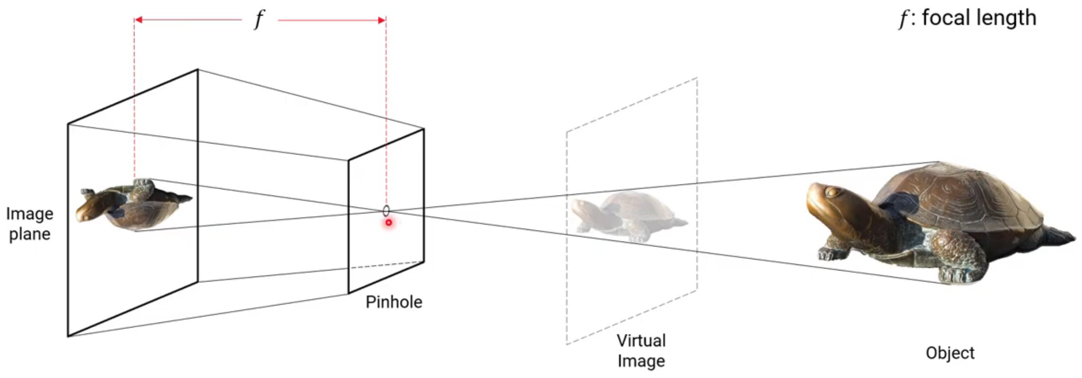


Figure 3.9: Pinhole camera model. Source: `http://prg.cs.umd.edu/`

As we will work with projective transformations we will move to homogeneous coordinates.

First, we consider a coordinate system with origin O_c and axes X_c, Y_c and Z_c (fig. 3.10). We will refer to this system as *camera coordinate system*. Then consider a rectangle in a plane perpendicular to the axis Z_c which has a common point (O_i) with the axis in the distance f from O_c. O_i will be the origin of so called *image coordinate system* with axes x and y parallel to X_c and Y_c respectively and with the same orientation.

Now considering point $P = (X_c, Y_c, Z_c, 1)^T \notin$ image plane, we can find its projection onto the image plane using similar triangles and obtain the projected point $p = (fX_c/Z_c, fY_c/Z_c, f, 1)^T$. We can ignore the f coordinate since it will remain the same for all points in the plane and obtain the points image coordinates $p = (x, y, 1)^T = (fX_c/Z_c, fY_c/Z_c, 1)^T$.

By this we found a linear mapping between camera and image homogeneous coordinates:

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX_c \\ fY_c \\ Z_c \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix}.$$

In practice, the origin of the image coordinates does not have to be defined at the point where the Z_c axis meets the image plane. It is often advantageous to define it for instance in the corner of the image. What then changes is that the point O_i has to

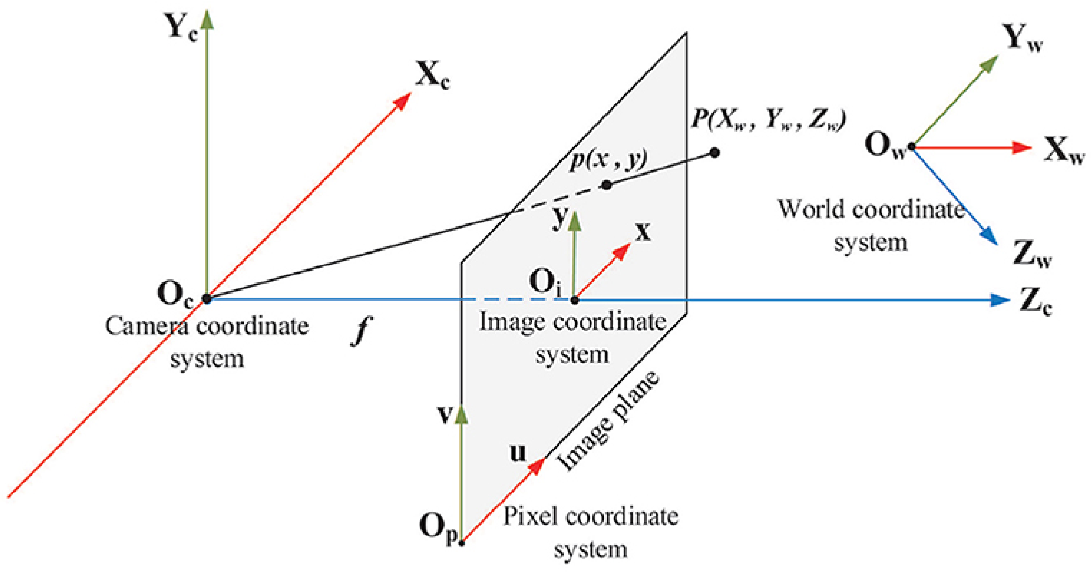


Figure 3.10: Different coordinate systems to deal with. Source: `https://www.frontiersin.org/articles/10.3389/fnbot.2020.616775/full`

be now expressed with respect to the new coordinate system (O_p, u, v) if O_i has new coordinates $(p_x, p_y)^T$, the expression of point p then changes slightly. If we for reasons described in [18] also consider possible skewed pixels skewed by the factor of s in the u direction and disproportionated in each direction by a different factor f_x and f_y, we obtain a new expression

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX_c \\ fY_c \\ Z_c \end{pmatrix} = \begin{bmatrix} f_x & s & p_x & 0 \\ 0 & f_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{pmatrix}, \tag{3.1}$$

If we now factor the mapping 3×4 matrix into two matrices:

$$\begin{bmatrix} f_x & s & p_x & 0 \\ 0 & f_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \overbrace{\begin{bmatrix} f_x & s & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix}}^{K} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The matrix K is called the *camera calibration matrix.*

We can now rewrite the expression (3.1) as $p = K[\boldsymbol{I}|0]P$, where $\boldsymbol{I}$ is 3×3 identity matrix and 0 is a 3×1 zero matrix.

The other matrix on the right seems to just be there so the dimensions work well. That is not really the truth.

Up until now, we have been expressing all the coordinates with respect to a coordinate system centred at the pinhole of the camera. This does not have to be so.

We wish to generalize to a *world coordinate system* with origin O_w and axes X_w, Y_w and Z_w. This is done through the workings of "the other" matrix.

We simply transform the whole space in such a manner that these coordinate systems become one. First we perform a translation which will result in $O_i = O_w$ and then rotate everything in such a manner that the axes of each of the systems are respectively equal.

Denoting the 3×1 coordinates of the camera in the world coordinate system as $\tilde{O}_c$, the affine 3×3 matrix performing the necessary rotations R, given a point in the world coordinate system, we can now represent it in the camera coordinate system:

$$X_c = \begin{bmatrix} R & -R\tilde{O_c} \\ 0 & 1 \end{bmatrix} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix},$$

where the matrix is 4×4 and 0 denotes $[0, 0, 0]$.

Plugging all this back into the equation (3.1), we obtain

$$p = K[R| - R\tilde{O_c}]P,$$

where P is now within the world coordinate system.

The product of the two matrices K and $[R| - R\tilde{O_c}]$ is called *camera projection matrix* and is usually denoted as P, that is

$$P = K[R| - R\tilde{O_c}].$$

The parameters of K are sometimes called *intrinsic* as it only carries information about the inner workings of a camera independently of the environment. Parameters of $[R| - R\tilde{O_c}]$ (often rewritten as $[R|t]$ for simplicity) are called extrinsic as they describe the camera only in the context of its environment and not itself.

These matrices can be obtained via camera intrinsic and extrinsic calibration described in [19].

Two cameras

The relationship between two cameras both described by their camera projection matrices, can be also captured by two matrices – Fundamental matrix F and/or Essential matrix E.

In order to obtain these matrices we need to understand the geometrical relationship between two different camera views, a point in space and its projections on both cameras.

Consider two different points C and C' as camera centres (pinholes) and two non-parallel planes I and I' not coincident with these points (fig. 3.11). If we now project a third point X in space first through C onto I and then through C' onto I', we obtain point x in I and x' in I'.

All the points mentioned so far lie in the same plane π called *epipolar* plane. All epipolar planes for various points constitute a pencil of planes with the line connecting C and C' as its common line, so called *base* line. Each epipolar plane intersects planes I and I' in so called *epipolar* lines denoted as l and l' respectively. These two also constitute a pencil in each respective plane with the intersection point of the base line with respective plane (e and e').

Epipolar line l' can also be described as a projection of the ray of point x in plane I. This means there should be a way of expressing line l' in terms of point x. The projective mapping which captures this relationship is described by a matrix F called fundamental matrix.

We know two points through which the line l' has to pass: e' and x'. In projective geometry this fact is captured by $l' = e' \times x' = [e']_\times x'$, where

$$[e']_\times = = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix},$$

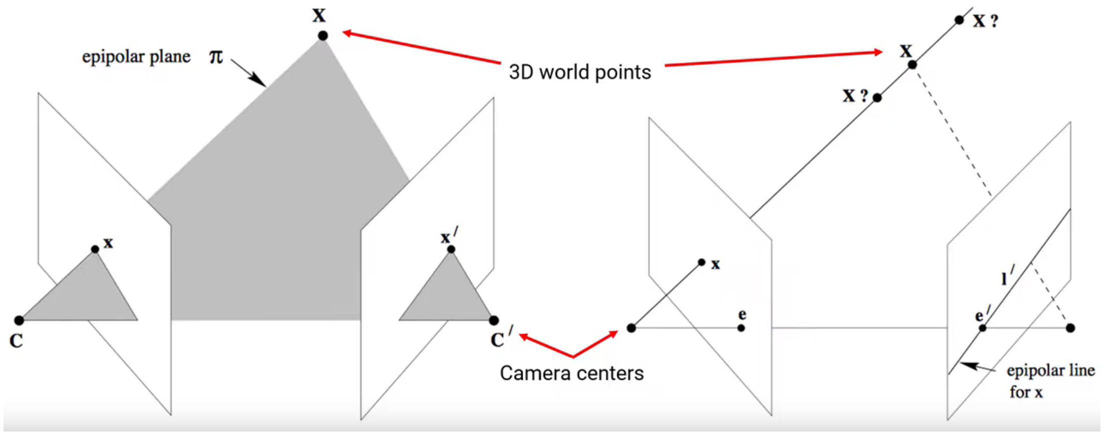


Figure 3.11: Epipolar model. Source: `https://towardsdatascience.com/introduction-to-epipolar-geometry-1bbe6e505b81`

if $e' = (a_1, a_2, a_3)$ (it can be verified that the second equation is true). Given a 3D-to-3D projective mapping H that takes point x to point x' as $e' = Hx$, we get $l' = [e']_\times Hx$. We define the fundamental 3×3 matrix as $F = [e']_\times H$ [16].

From the fact that x' lies on $l' = Fx$, it follows that

$$0 = x'^T l' = x' F x = 0.$$

The last equation is called *correspondence condition.* It gives us a way of describing fundamental matrix without reference to the camera matrices.

This leads to the Linear 8-point algorithm which finds (estimates) the fundamental matrix from 8 points with their correspondences in the other image. A detailed account of this method can be found in [16].

From investigating some further properties of this matrix we can find out that it only has 7 degrees of freedom. F is also unique for the given pair of cameras and has always rank 2 (therefore $\det F = 0$) and therefore not invertible. These findings represent certain constraints that can be used when finding the values of F.

An important result ([20]) is that F can be decomposed into camera projection matrices P and P' of individual cameras. This decomposition however is not unique, that is there is a whole set of solutions which are same up to a projective transformation.

If we know calibration matrices of the cameras, we can rewrite two corresponding points x and x' as $\hat{x} = K^{-1}x$ and $\hat{x}' = K'^{-1}x'$ (normalized camera coordinates), which will still be two corresponding points. We can derive that

$$x'^T F x = K'^T \hat{x}'^T F K \hat{x} = \hat{x}'^T \underbrace{K'^T F K}_{E} \hat{x} = 0.$$

We will call the matrix

$$E = K'^T F K$$

the essential matrix of a pair of images.

From the properties of F, it can be proven that $E = [t]_\times R$. It has 5 degrees of freedom, its rank is zero.

Since we express the camera position in canonical terms of the first camera, we can see that $P = K[I|0]$ and $P' = K'[R|t]$. We observe the parameters R and t in the essential matrix. If we can decompose it and extract these information, we would obtain the relative position of our cameras. If we do so we obtain four possible solutions for P'. From these only one is physically correct, that is the object is in front of both cameras.

Our path in this section so far led to obtaining parameters of each camera and their relative position. Now it is time to localize actual points of our model.

For this we will use the linear triangulation method (LTM). In each image we have several matched points and from earlier reconstructions, we should have camera projection matrices for each image.

Point X should then satisfy equations $x = PX$ and $x' = P'X$. Each of these equations if defined up to a certain scale λ that is it is also true, that $\lambda x = PX$. This unknown scale factor is eliminated by taking a cross product of both sides:

$$\begin{aligned}(\lambda x) \times PX &= PX \times PX \\ \lambda(x \times PX) &= 0 \\ x \times PX &= 0\end{aligned}$$

This gives rise to three equations

$$\begin{aligned}u(p^{3T}X) - (p^{1T}X) &= 0 \\ v(p^{3T}X) - (p^{2T}X) &= 0 \\ u(p^{2T}X) - v(p^{1T}X) &= 0,\end{aligned}$$

where $x = (u, v, 1)$ and p^{iT} are the rows of P, two of which are linearly independent.

From here and equation of the form $AX = 0$, where

$$A = \begin{bmatrix} up^{3T} - p^{1T} \\ vp^{3T} - p^{2T} \\ u'p'^{3T} - p'^{1T} \\ v'p'^{3T} - p'^{2T} \end{bmatrix}$$

and $X = [x, y, z, 1]$, can be formed for each pair of corresponding points in the images.

For known calibration matrix (using E), this process yields results that are all mutually similar (same up to scaling, rotation and translation).

For unknown calibration matrix (using F), the results are subjected to projective ambiguity.

These ambiguities can be removed by stratified reconstruction in which a projective reconstruction (obtaining P and X up to an unknown projective transformation) is refined into an affine one and even to a metric one, where each step requires some additional information.

The step between projective and affine reconstruction is realized via mapping all the reconstructed points through matrix

$$H = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 \end{vmatrix},$$

where $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$ is a plane in the projective space which H maps to $(0, 0, 0, 1)^T$. It is obtained as $\pi = (v_1 \times v_2) \times (v_2 \times v_3)$, where v_i are intersection points of three known pairs of parallel lines of the model.

Using a process called Cholesky factorization we can obtain a matrix for getting rid of affine distortions. This process is rather lengthy, it can be found in [21].

For problems that come with adjusting images for which K is unknown, most software requires some knowledge of camera calibration.

Tens, hundreds, thousands and more cameras

Perhaps surprisingly, most of the processes described in the previous section and their outcomes were actually already executed within the frame of RANSAC hidden behind the veil of ”computing model parameters using sampled data points.” But now we have all the information we need to execute the core part of SfM. Here, the process will be fairly simple.

There are three main paradigms of SfM : incremental, hierarchical ([22]) and global ([23]).

Here, we will show the algorithm of the incremental way, the name of which basically gives away the method we will use.

The image pair we will use to initiate the reconstruction (*seed pair*) is chosen amongst all the pairs found in previous steps (for reasons described in [24], we do not want this pair to differentiate only by a rotation) based on two criteria:

1. It has the highest possible number of matches, while
2. being well interconnected with other images.

The first criterion allows us to reconstruct a large amount of points while being sure that our model is very accurate. The second criterion ensures there will be more accessible images to match with the initial ones to minimize the chance of this pair being isolated from the rest of the reconstruction.

Knowing the relative positions of the cameras C and C': $P = K[I|0]$, $P' = K'[R'|t']$, we choose the initial scale (we already mentioned that all reconstructions with known K yield mutually similar models) by setting $||t'|| = 1$.

Then we reconstruct all the matching points using triangulation, after which we apply bundle adjustment (below) to refine camera matrices and 3D points.

Adding a new camera view C'' is done by choosing another image that has common key points with both the previous images. Like that we can find some points that we already reconstructed in 3D in the new view. Now again by utilizing the equation (3.?), this time knowing the coordinates of point X, we can find the matrix P'' of the new view with respect to the reconstruction done in the previous step.

From here we can reconstruct new points that are shared only by cameras C and C'', or C' and C''. With these new points we can iteratively repeat this process and reconstruct all the 3D scene’s key points common to atleast a pair of images in the original image set. The process is sketched in figure 3.12.

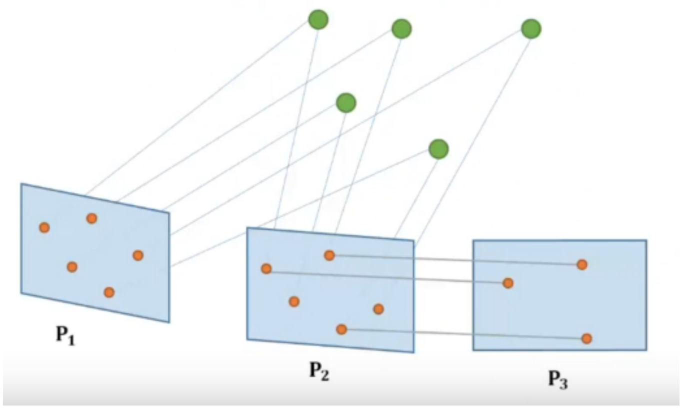


Figure 3.12: Once we find some matching points, we can reconstruct them in other images. Source:`https://www.comp.nus.edu.sg/~leegh/`

This process however is burdened with errors caused by various reasons, such as camera distortions, inaccuracies in point matching from this following fact that parameters of cameras and point reconstruction are only estimated through the above algorithms (pinhole camera itself is only an idealized model).

The step of bundle adjustment aims to minimize these errors.

The idea of bundle adjustment is in the fact that each reconstructed 3D point X_j, as it was only estimated from an image point x_{ij} (that is point projecting point X_j from camera pose P_i), can be projected back onto the original image to a point $\overline{x}_{ij}$ that is naturally not the original point from which X_j was constructed. Bundle adjustment aims to minimize the Euclidean distance between the original point x_{ij} and this newly obtained point $\overline{x}_{ij}$ for all points across all views. Mathematically it is an optimization problem aiming to minimize the square distance between these points, it can be expressed as [18]:

$$\arg\min_{P,X} \sum_i \sum_j ||x_{ij} - \overline{x}_{ij}||^2,$$

where $\overline{x}_{ij} = \frac{P_i X_j}{\hat{x}_3}$, where $\hat{x}_3$ is the third element of $P_i X_i$. Division by $\hat{x}_3$ normalizes $\overline{x}$.

There are several algorithms based on non-linear search for solving this problem, such as Newton's method or Levenberg-Marquardt algorithm ([25]).

Based on bundle adjustment, points and cameras of the reconstruction can be adjusted or fully removed from the reconstruction if they show hight levels of error.

Iterative method is advantageous if we want to enhance the reconstruction by adding new images to the view later on. The whole model does not have to be recomputed, only parameters of the new camera are extracted and it cane be quickly added little demand for computational time.

After all these procedures, we are left with what is called *sparse model* of the scene with camera poses (fig. 3.13). A one more step can be taken here in order to obtain a *densemodel* (fig. 3.13). The dense model is way more suitable for further processing such as meshing. New points are obtained based on image depth maps estimation, which is yet another important branch of computer vision. An efficient way of doing this is Semi-Global Matching (SGM) described by Hirschmüller in [26].

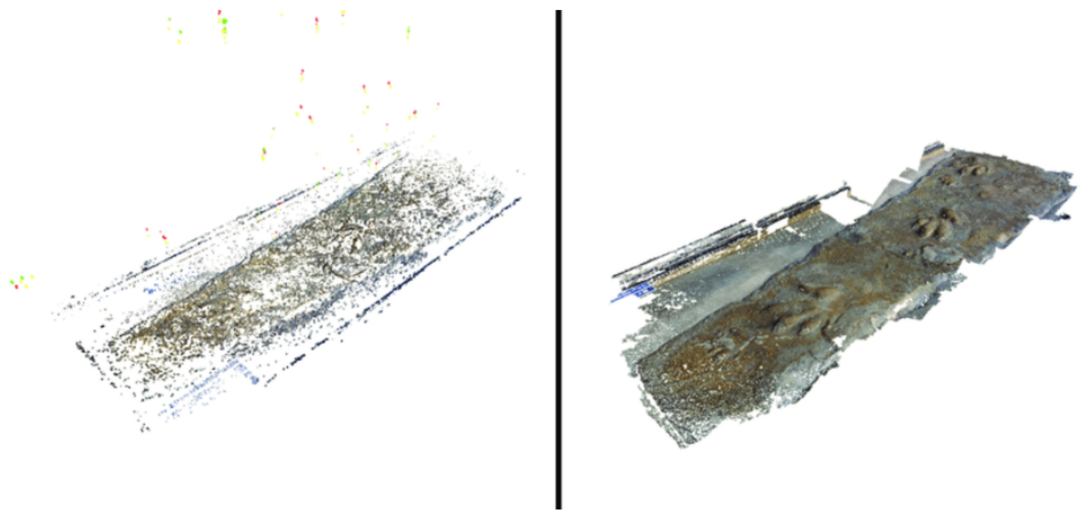


Figure 3.13: Sparse and dense model obtained by photogrammetry. Source: https://www.researchgate.net/figure/Sparse-point-cloud-generated-by-Bundler-Green-red-and-yellow-points-indicate-camera_fig2_225683089

For our purposes we will assume the sparse reconstruction was exceptionally successful and stick with it.

3.3 After SfM – Meshes anew

Previous steps left us with a cloud of points. We would however want to create a dense geometric surface representation of the scene.

This is achieved by triangulation. However, unlike in the case of triangulation described above, here we do not have the points parametrized over a (u, v) grid, we only have their coordinates in 3D space.

Generally, solving the problem of meshing a point cloud is a hard problem, even more so in the case of photogrammetry where the modelled objects can significantly vary in their geometries. For these reasons a plenty of methods, such as projection-based methods, are not suitable. Most common methods here are implicit and explicit methods (such as Delaunay methods) ([27]).

We will take a brief look at Delaunay methods. As the name suggests these methods utilize the Delaunay triangulation and tetrahedralization.

Delaunay triangulation is a triangulation process of a set of points aiming to maximize the minimal angle in each triangle. This is done by ensuring that every triangle's circumcircle is empty, that is it does not contain any other point from the set that is not a vertex of the triangle.

An important term regarding Delaunay triangulation is Voronoi diagram. It is a dual graph of the Delaunay triangulation, that is if every face of the triangulation is mapped to a point of the Voronoi diagram, every point maps to a face and edge to an edge. Voronoi diagram forms a proximity cell $(V(s))$ around each vertex of the triangulation, that is cell defined as

$$V(s) = \{x \in \mathbb{R}^2 : ||x - s|| \leq ||x - t||, \forall t \in S\},$$

where S is the set of all vertices of Delaunay triangulation (fig. 3.14) .

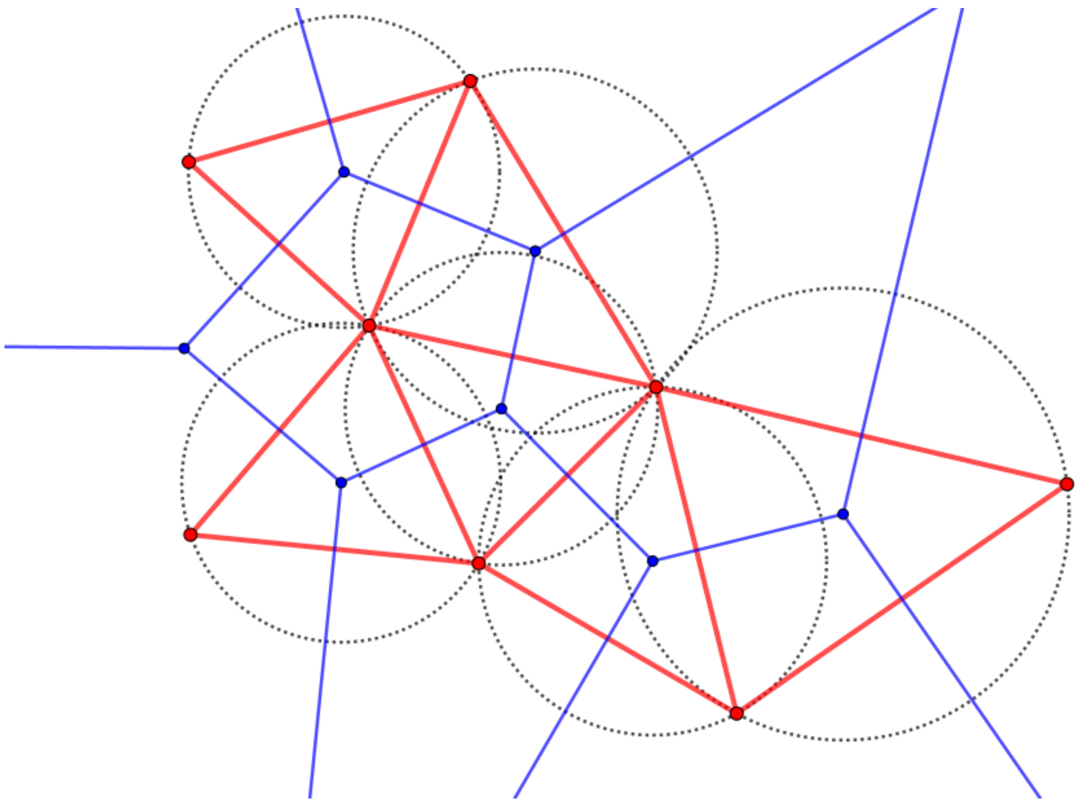


Figure 3.14: Delaunay triangulation (red) and its dual Voronoi diagram (blue).

This idea can be reproduced in higher dimensions as well. Triangles become tetrahedra, circumcircles become circumspheres and planar Voronoi diagram becomes spatial.

One of early algorithms using Delaunay triangulations and Voronoi diagrams for surface reconstruction, introduced in [28], is called Crust.

It operates on a relatively simple premise: First a 3D Voronoi diagram for the obtained point cloud (S) is computed. For each sample point s its two *poles* are identified as the two extremal (farthest) vertices of the Voronoi cell $V(s)$ on each side. We call the set of all poles P. Now, we perform Delaunay tetrahedralization of the set $P \cup S$. From this tetrahedralization, we discard every triangle which has a point from P as its vertex. What we are left with is a set of triangles with points from S as its vertices forming so-called *crust*.

If the point cloud is sufficiently dense, the crust is a good reconstruction of the desired surface. This procedure exploits the fact that the Voronoi cells of points on the surface are elongated in a direction perpendicular to the surface. From this, the extremal vertices of these cells are a good estimate of the medial axis of the surface. The medial axis separates opposite patches of the surface, therefore the Dalaunay triangulation of $S \cup P$ can never connect two vertices on opposite patches, there is always a pole in the way.

This basic idea was enhanced over time and crust still proves to be a very robust method for meshing. A closeup of a photogrammetry model obtained by Delaunay method is in fig. 3.15.

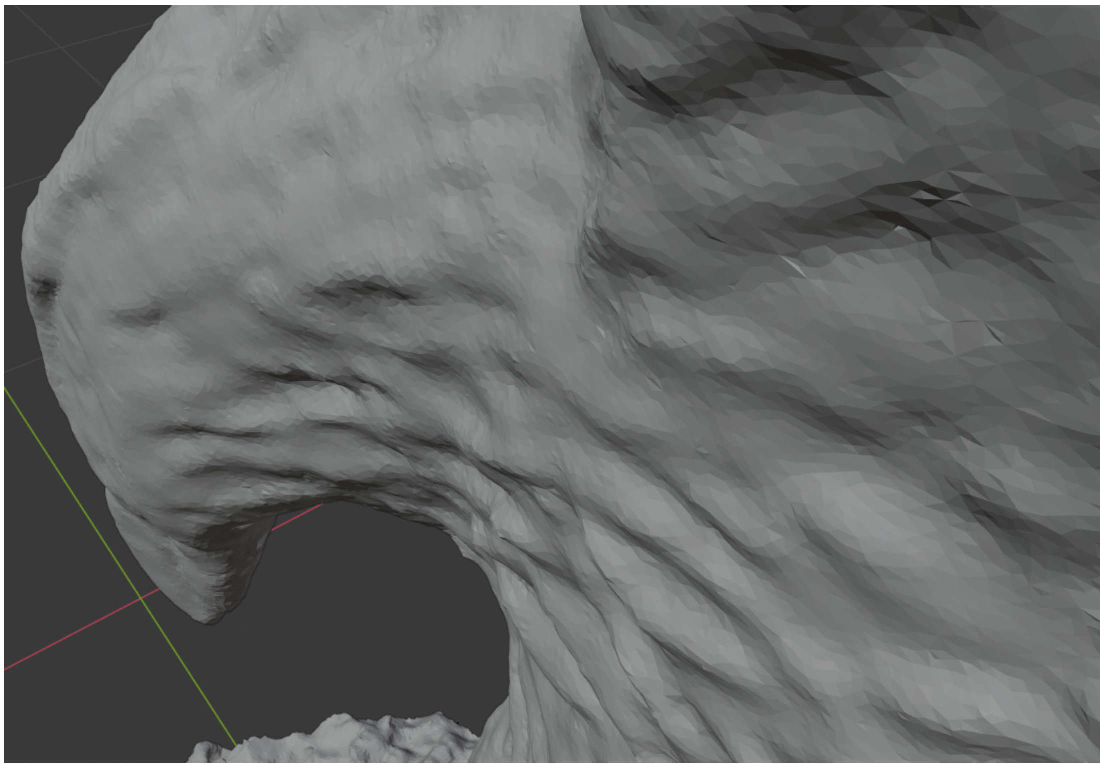


Figure 3.15: If the image set was good enough, the density of points and triangles is very high.

A comprehensive overview of other Delaunay methods can be found in [27] and a closer look at one specific Delaunay method based on voting procedure which is very commonly used can be found in [29]. Some very recent methods for large data clouds are described in [30] and [31].

When this initial meshing is done, there is still a clean-up to be done. No point cloud is perfect and may still contain noise artefacts and the meshing itself can locally produce undesired geometries. These can be removed by applying Laplacian filtering as described in [32].

3.4 Complex 3D models fast and easy

In this section we will specifically show a way of obtaining a 3D digital model through photogrammetry using Meshroom and Blender. We will use an image set from https://support.pix4d.com/hc/en-us/articles/360000235126-Example-projects-real-photogrammetry-data#label5 so our results are reproducible (fig. 3.16). Meshroom can usually handle even users own image sets taken with a mobile phone.

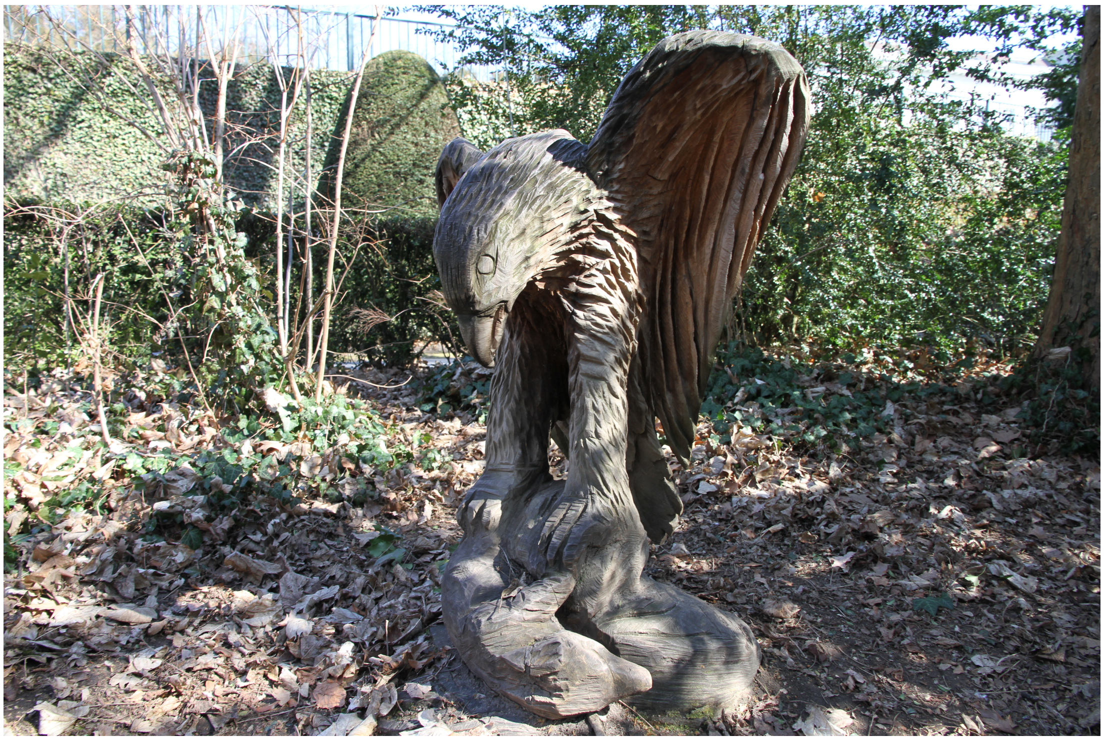


Figure 3.16: Example image from the chosen image set. The image is very sharp and camera settings do not vary between images.

Some basic rules for image acquisition are:

- All images should be as sharp as possible. Fast shutter speed, reduced ISO and aperture can help with that. Do not change the camera settings throughout acquisition.
- Do not move the object or things around it in the scene. Try to not change the lighting too much either.
- Avoid large textureless smooth surfaces, very glossy objects, transparent objects.
- Take images from all possible angles.

If the reconstruction does not go well, it is possible to add new images later to augment it.

3.4.1 Meshroom

We use Meshroom 2021.1.0 (`https://alicevision.org/`).

The workflow in Meshroom is actually very easy and it has a simple environment (fig. 3.17).

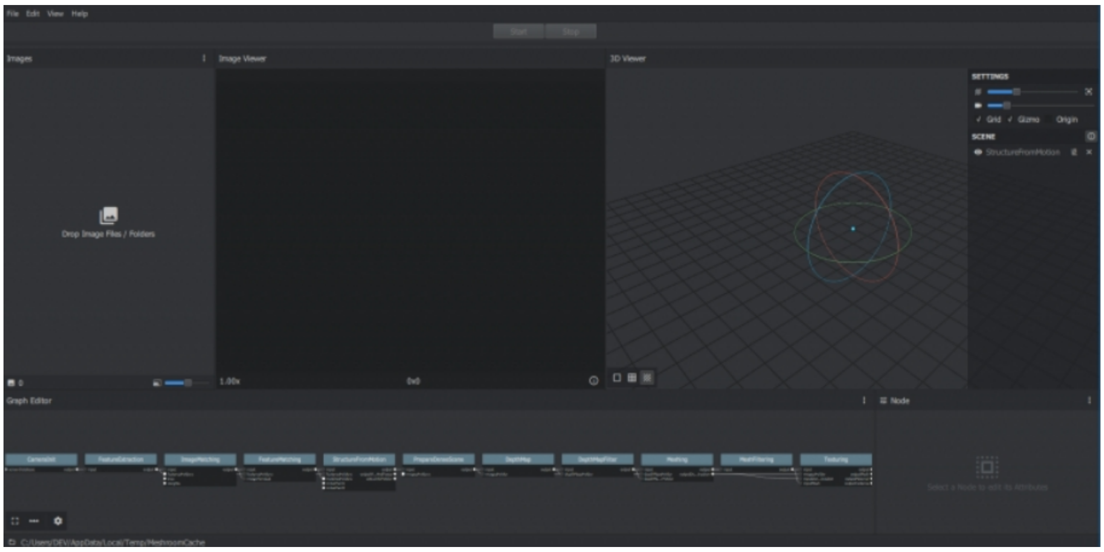


Figure 3.17: The environment is simplistic, the image set is displayed on the left, in the middle we can view a specific image, on the right the model will be reconstructed and on the bottom the program's pipeline is displayed.

All we need to do to initiate the process is dropping a set of imaged to the left window and press Start on the top of the screen. The reconstruction can take rather long even on machines with good CPU and GPU depending on the image sample size. The progress of the process is highlited in the bottom pipeline menu (fig. 3.18). Advanced users may alter the pipeline to obtain some variations of the final result. It may be interesting to observe the difference in processing times if individual steps.

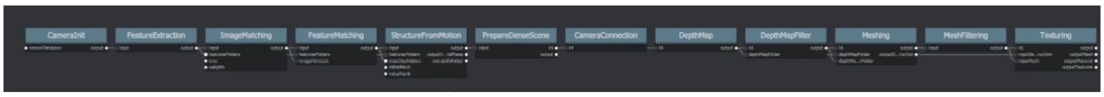


Figure 3.18: Notice that the pipeline pretty much follows the path we took in the text above.

First thing displayed will be a point cloud with camera positions (fig 3.19).

Once the program runs through the whole process, the object is displayed meshed and textured (fig. 3.20).

Once the object is done, it can be moved to a CAD program and further processed. As its final act, Meshroom saves the scene as an .obj file, which is readable for most 3D software. In a software the object is added as a mesh for further work. Then it can be rendered (fig. 3.21).

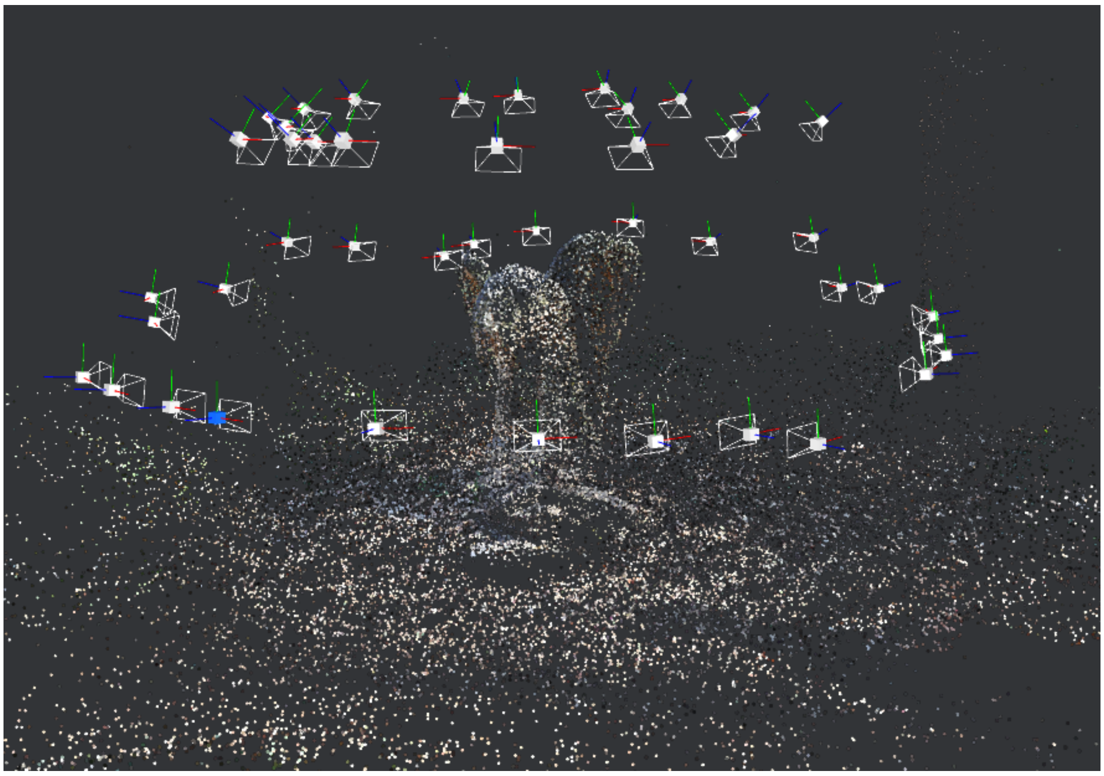


Figure 3.19: Sparse reconstruction of the scene.

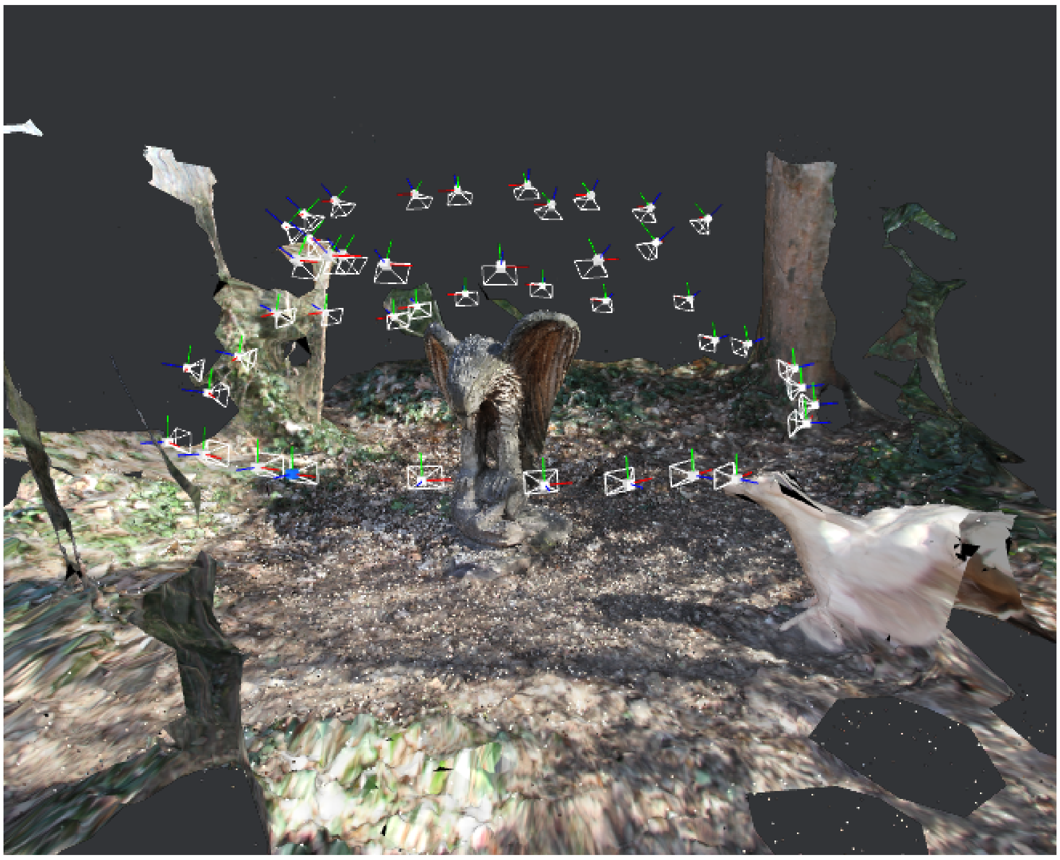


Figure 3.20: Scene done.

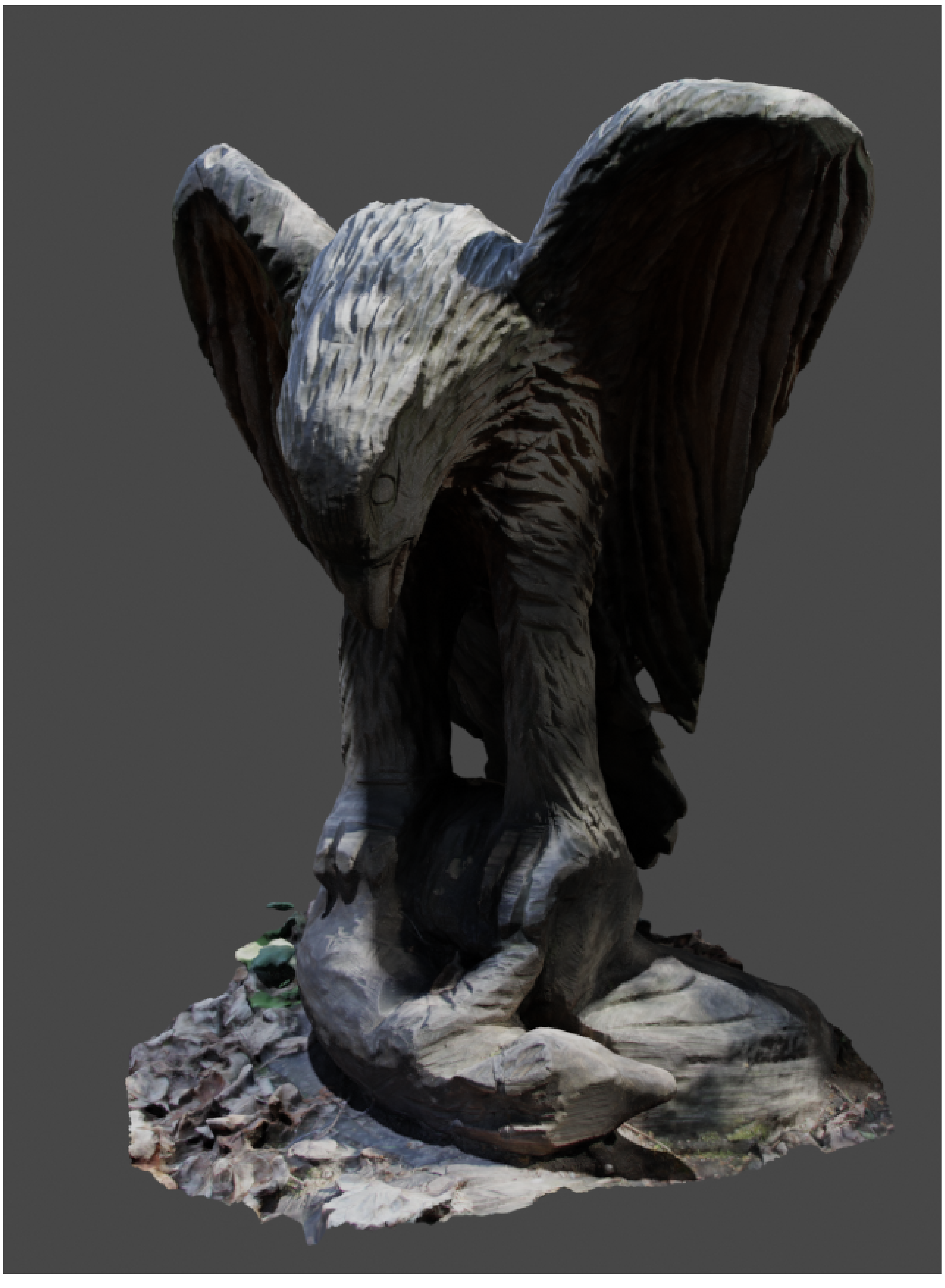


Figure 3.21: Most of the scene got cut off and the statue is rendered.

4. 3D printing

With its roots going as far as early 1980s when its concept was developed by an American inventor Charles W. Hull, 3D printing has experienced an unprecedented rise in prominence over the past few years as a relatively cheap and easy to use means of manufacturing.

3D printing is an umbrella term for several techniques of additive manufacturing (AM) which, unlike the usual manufacturing processes, creates objects by adding or joining material (usually layer by layer) rather than whittling it away. This not only saves material costs but also allows for a wider scale of complex geometries which can be produced [33].

3D printers (fig. 4.1) usually require input from a CAD/CAM program in form of a 3D model or preprocessed data obtained by reverse engineering. The bridge between a CAD software output and a 3D printer is usually an .stl (stereolitography) file processed by a slicing software and translated to g-code (a widely used computer numerical control (CNC) programming language) which then controls the behaviour of the printer itself [34].

Figure 4.1: FDM 3D printer. Source: `https://www.prusa3d.cz/`

4.1 Preprocessing

We already mentioned, there are some steps that need to be taken before the 3D printing can take place. In previous chapters, we have shown some methods through which we can obtain a digital 3D model.

This model now needs to be processed in such a manner that it can be understood by a printer. The way in which we can mediate the communication between a CAD program and a printer is sketched above. In this chapter we will dive a little bit deeper.

We start with a more theoretical part. Most printers work on the basis of decomposing a model into individual slices or layers which, stacked atop each other form the object.

4.1.1 The theory that allows us to print anything and the reality that ruins it

The time comes to question the whole process sketched below. Is it really possible to print any real life object as a system of layers? How do we know that?

The answer to this problem comes in a form of a theorem formulated by an Italian mathematician Guido Fubini (1879 – 1943) in 1907.

Theorem 1 (Fubini). *[35] For $p, q \in \mathbb{N}$ and $p + q = n$:*

1. *If*
$$\int_{\mathbb{R}^p} \left(\int_{\mathbb{R}^q} |f(x,y)| \, dy \right) dx < \infty$$
or
$$\int_{\mathbb{R}^q} \left(\int_{\mathbb{R}^p} |f(x,y)| \, dx \right) dy < \infty,$$
then $\int_{\mathbb{R}^n} f(x,y)dydx$ converges and

2. *if $\int_{\mathbb{R}^n} f(x,y)dydx$ converges, then it always holds that*
$$\int_{\mathbb{R}^n} f(x,y)dydx = \int_{\mathbb{R}^p} \left(\int_{\mathbb{R}^q} f(x,y)dy \right) dx = \int_{\mathbb{R}^q} \left(\int_{\mathbb{R}^p} f(x,y)dx \right) dy.$$

To proof this theorem, we would require additional background information of measure theory which is beyond the scope of this work. Proof of a more general version of this theorem can be found for instance in `https://www.math.cuhk.edu.hk/course_builder/1617/math5012/RealAnalysis_Chapter_7.2017.pdf`.

The important result of this theorem for our purposes comes when we substitute $n = 3$. Then we get to our 3-dimensional space and see that $p = 2$ and $q = 1$ or vice versa. This in language of mathematics says that any 3-dimensional real world object (for such an object conditions in Theorem 1 – 1 always hold) can be described in terms of a one-dimensional spectrum of two-dimensional shapes, that is layers.

As it tends to be, pure mathematical concepts meet their shortcomings when faced with reality. In the same fashion the Fubini's theorem has to be taken with a grain of salt when applied to 3D printing as there are limits when it comes to slicing resolution physical stability of a model, physical properties of printing materials and feasible printing times. More detailed information and some of other challenges in 3D printing to date, primarily fused deposition modelling, can be found in [36].

4.1.2 Slicing the (virtual) matter

STL format is currently the most common format to represent 3D models in additive manufacturing. In an ASCII STL file, a 3D model is triangulated and represented using the x-, y- and z-coordinates of vertices of its triangular facets and unit vectors of their normals. Facet normals are included to capture the orientation of the model. Most of the recent slicing programs are able to correctly estimate the orientation even without facet normals which is reflected in some stl exports from programs like Mathematica which do not waste computational time and fill the facet normal line with zeros.

For better accessibility some extra information such as model name, facet normal and vertex is included in an stl file.

Now, we will proceed to the slicing of a model itself. There are two possible ways of slicing a model when it comes to the direction of slicing: unidirection slicing and multidirection slicing. Unidiretion slicing algorithm performs slicing in one direction along the build-up direction. It is the more common method for its simplicity and easiness of implementation. Its big disadvantage which is tackled by multidirection slicing is that it does not respect the geometry of the model so overhang parts often need supports. Building and consequently removing supports in post-processing can be very material and time consuming especially when it comes to mass production and production of big parts.

For multidirection slicing the printer has to be able to deposit material in several directions which is not the case for most printers. Nevertheless the prospects of saving material and time led to development of several algorithms which can perform multi-direction slicing ([37]). The problem of these algorithms is they are not universal and work only on certain types of geometries.

For purposes of this thesis we will focus on unidirection slicing.

When it comes to layer thickness, there are again two ways to go – uniform and non-uniform slicing. The differences between these two pretty much parallel the differences between uni- and multi-direction slicing. That is uniform slicing is more wide-spread as it is easier to compute and is supported by most machines. Non-uniform slicing can save material and time as it adjusts depending on the model's geometry which can again cause problems when it comes to its universality.

Slicing algorithm

Unidirection slicing (fig. 4.2) algorithms slice the model into a variety of 2.5D layers[1] of uniform thickness parallel to the build direction. The accuracy of the printed model is influenced by the said thickness of the layers – the thinner the layer, the more accurate the model is. The accuracy however has to be weighted against the increased time required for printing and technical possibilities of both filaments and printers.

With bigger .stl files, efficiency of slicing algorithms becomes a concern. In this area, major advancements were made. As an example we will briefly present a tolerant slicing algorithm as presented in [38]. The advantage of this algorithm is that it solves the memory bottleneck problems of large .stl files by reading one facet at a time. If the facet intersects with the cutting plane, it is processed and subsequently disposed which frees up a lot of computer memory.

As can be seen in the figure of the algorithm's flowchart 4.3, the algorithm operates in two stages. The first is pre-processing of the model. In this stage the algorithm finds an optimal orientation of the part with regards to such things as printing time, surface

[1]2.5D refers to the fact that despite representing a planar situation, these layers also have thickness

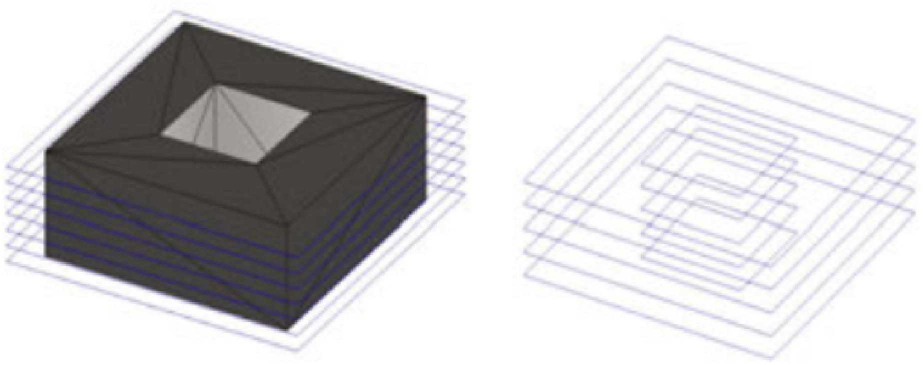


Figure 4.2: .stl model with slicing layers (left) and resulting binding polygons (right). Source: [37]

quality, support structures required and other optional criteria. The second stage is where the slicing itself is performed.

In the slicing stage the z-coordinates of vertices of facets are compared to that of the cutting plane, when an intersection is found, a line segment is constructed and subsequently joined in a form of a list with other line segments in the layer. Facets that do not intersect the cutting plane are ignored.

An important part of this operation is the bound tolerance. That is the way in which it is decided whether a facet should be sliced. Every structure in computing is burdened with a numerical error since there is a limit to the digits it is worth remembering for the program. When the difference between two values is smaller than a set tolerance, they are assumed equal. Finding and defining the line segments based on how many of the facet's vertices lie within the range of the cutting plane's z-coordinate $\pm$ the bound tolerance is discussed into detail in the original work ([38]) and is not important within the scope of this thesis.

The pointing directions of the contour line segments created during the slicing process are determined by the orientation of the model (mostly considering the normal orientation) and are used for subsequent connection of said segments in a slice contour.

This algorithm shows good stability even when dealing with faulty .stl models with missing or degenerated facets [37], [38].

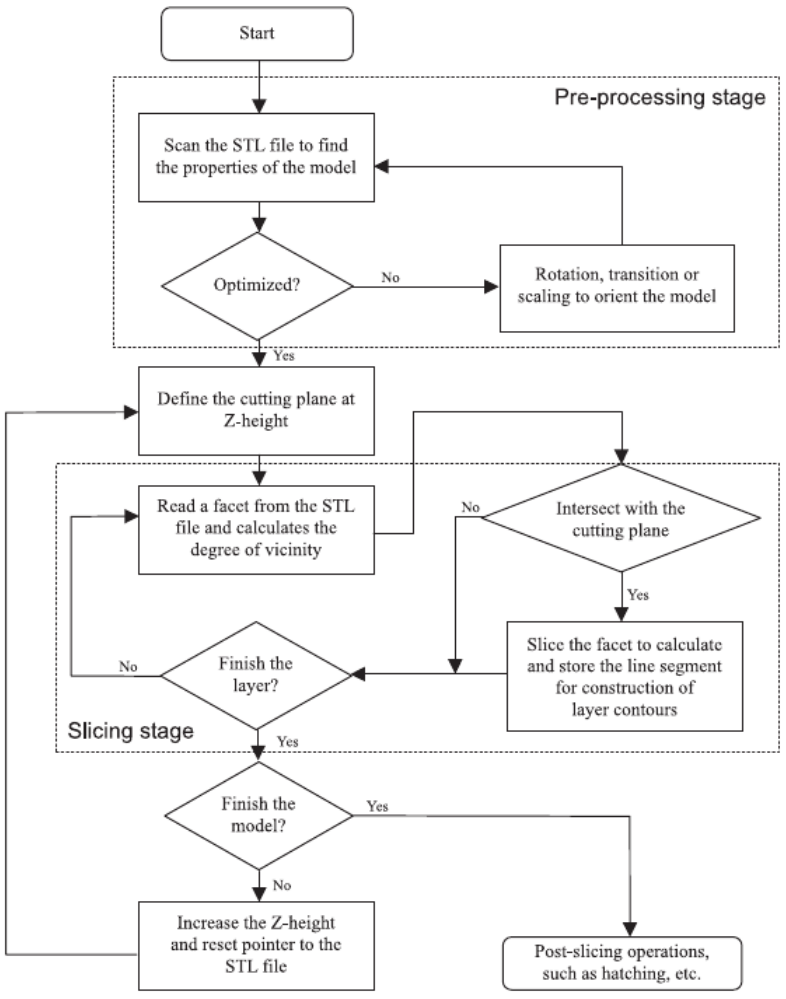
4.1.3 2D path planning

The next step after slicing an object and before its final fabrication is path planning for printing individual layers. This process has many constrains laid upon it such as surface quality, shape accuracy, infill distribution quality, time of printing, material properties, amount of material use and many more. A detailed review of literature on this topic can be found in [39].

Problem formulation and ways to go about it

We focus on a 2D path planning problem corresponding to printing a slice of a given object. We try to find the most efficient way to "draw" the given pattern under given constraints. When printing one layer, the nozzle can move in x- and y-direction within the given area and draw continuous curves.

Different paths show different qualities and their use can be beneficial depending on the given constraints. Mathematically, it is a linear programming problem.



Start
Pre-processing stage
Scan the STL file to find the properties of the model
Optimized?
No
Rotation, transition or scaling to orient the model
Yes
Define the cutting plane at Z-height
Read a facet from the STL file and calculates the degree of vicinity
No
Intersect with the cutting plane
Yes
Slice the facet to calculate and store the line segment for construction of layer contours
No
Finish the layer?
Slicing stage
Yes
Finish the model?
Yes
No
Increase the Z-height and reset pointer to the STL file
Post-slicing operations, such as hatching, etc.

Figure 4.3: A tolerant slicing algorithm flowchart. Source: [38]

Raster path (fig. 4.4) is based on planar ray casting along a given direction. Strategy fills the region with a set of scan lines with finite width. It is commonly used due to its simple implementation and suitability for various boundaries.

Grid path (fig. 4.4) shares the ideological basis with the raster path but fills the region with two not necessarily perpendicular rasters. This method is usually more time consuming but improves some of mechanical properties of the product.

Zigzag path (fig. 4.4) is the most common method. It is derived from the raster path but reduces the number of tool-path passes by combining the parallel lines into a continuous path. All three paths mentioned so far show poor results when it comes to outline accuracy.

Contour offset path (fig. 4.5) addresses the problem of the three former paths by following the geometry of the boundary contours. This method can be slightly more time consuming and harder to implement for different geometries.

Spiral path (fig. 4.5) is widely applied in CNC machining. It also solves the problem of outline accuracy but it is only suitable for special geometrical models.

In order to achieve better results, some of the attitudes above can be combined in so-called **hybrid paths**. A widely used path pattern in many 3D printers combines contour lines approach for the outer contours and zigzag path for the infill. A detailed account of such approach using zigzag path combined with contour offset NURBS-based paths can be found in [40].

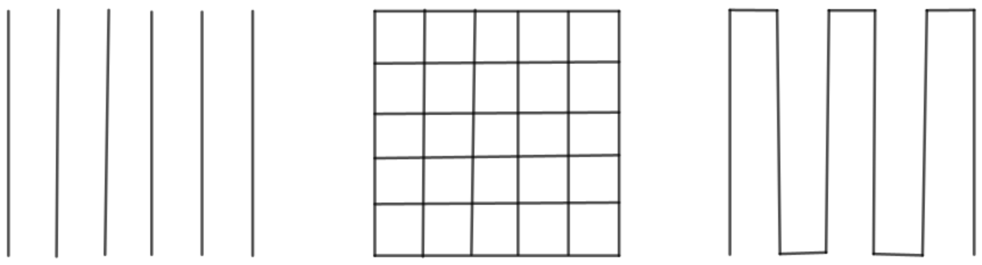


Figure 4.4: Raster path (left), grid path (middle) and zigzag path (right)

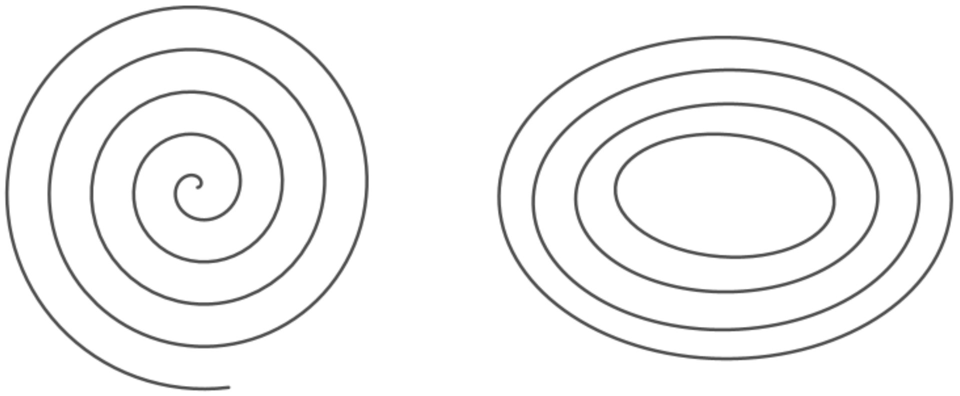


Figure 4.5: Spiral path (left) and offset path (right)

4.2 An overview

This chapter will be mainly dedicated to the currently cheapest, most popular and common method of 3D printing feasible for even personal 3D printers – fused deposition modelling (FDM) using thermoplastic polymers. However, in this section we also include information about some other possible methods and materials used for 3D printing processes in general. Namely we will mention three common techniques for AM other than FDM – binder fusion, powder-bed fusion (PBF) and stereolitography (SLT) techniques. There are many more such as contour crafting, allowing us to print even whole houses, direct energy deposition, which is somewhat of a hybrid between SLM and FDM used mainly on metals and alloys and plenty other techniques hidden behind the '3D printing' term.

4.2.1 Methods

Fused deposition modelling

FDM is a commonly used 3D printing method during which a continuous filament of thermoplastic polymer is heated to the point of reaching a semi-liquid state and is consequently extruded on a platform or atop previous layers to create the desired object. The thermoplasticity is an important property of the polymer which allows it to adhere to other layers and solidify at room temperatures. The semi-liquidity of the polymer is also its weakness since the material usually collapses without some kind of supporting structures. Depending on the material and intended use, post-processing may be needed.

FDM allows the user to easily produce cheap products at a relatively high rate on cheap machines which made FDM printers the current leading technology when it comes to DIY (do it yourself) printing.

The weakness of these printers lies primarily in the materials suitable for them, which are fairly limited and commonly show weak mechanical properties. Without later post processing, objects printed on these printers also tend to have an obvious layer-by-layer appearance which aside from aesthetic problems may also cause distortions, so a big deal of post-processing is often needed.

For the many problems mentioned above, this method is often not the first choice for many industries. It is most notably used for manufacturing things like toys and composite parts. For its price, speed and versatility it is often utilized for rapid prototyping and visualisations.

Binder jetting

Binder jetting (fig. 4.6) is a method during which a droplets of suspension (binder) are delivered by a print head onto a thin layer of substrate which it causes to solidify by connecting the grains of the substrate until it is able to hold another layer. This process is usually fairly fast depending on the interaction between the binder and substrate which adds flexibility in regards of printing even complex geometries. The substrate provides enough support for the object so no additional structures are needed. After all the layers are done and the suspension has dried, the object has to be subjected to some post-processing in form of removing excess material ([41]).

This method is suitable for fast printing of large very parts in large volumes which makes it perfect for various industries, but in recent years some major developments in precision for small scale objects were made. What makes this method even more interesting is the gamut of printing materials. Very common material is ceramics, but even some metals, polymers and even materials like sugar can be processed using this method.

Possible drawbacks are problems with adhesion between layers, distortions of lower layers caused by the weight of the material above, lower print quality, coarse resolution and errors caused by post-processing such as erosion.

Binder jetting finds its use primarily in heavy industries producing large parts, building construction but also smaller scale fields such as medicine and electronics.

Powder-bed fusion

PBF (fig. 4.7) is also amongst the more wide-spread methods of 3D printing. At first, a thin layer of fine powder is evenly spread on build platform, a laser or an electron beam then selectively melts or sinters the powder's particles together at pre-specified points. Once a layer is done, the process is repeated until the desired 3D structure is attained. The excess powder that was not melted supports the object, therefore no support structures need to be built like in the case of FDM. The unbound powder has to be removed however, which can be an uneasy task for highly complex geometries.

Powder-bed fusion excels at resolution, quality and versatility of materials that can be used. Selective laser sintering (SLS) can be used on a variety of materials, such as polymers, alloys and metals, whereas its 'partner in crime' selective laser melting (SLM) shows great results when working with certain types of metals.

Major drawbacks of this method which prevent its wide spread on the level of FDM printers are slow speed, high costs and energy consumption.PBF finds its applications

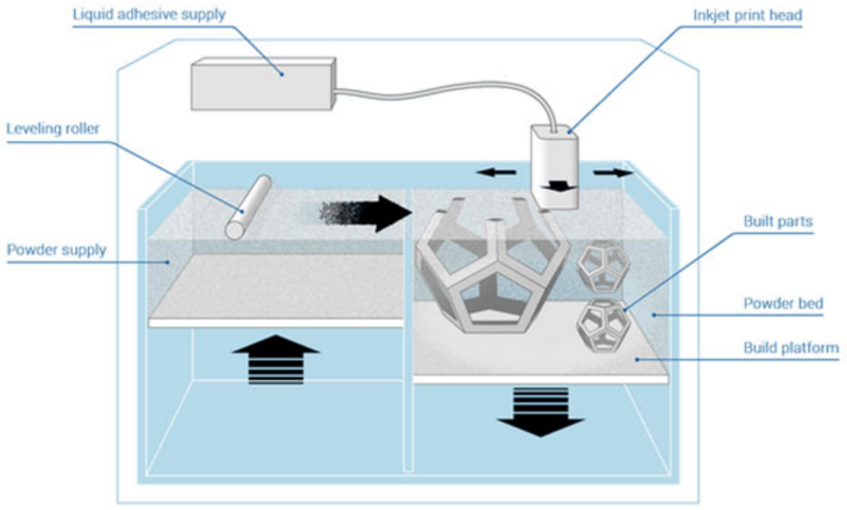


Figure 4.6: Working principle of binder jetting. Source: `https://www.3dnatives.com/en/powder-binding100420174/`

in a widening area of fields such as biomedicine, aerospace and electronics.

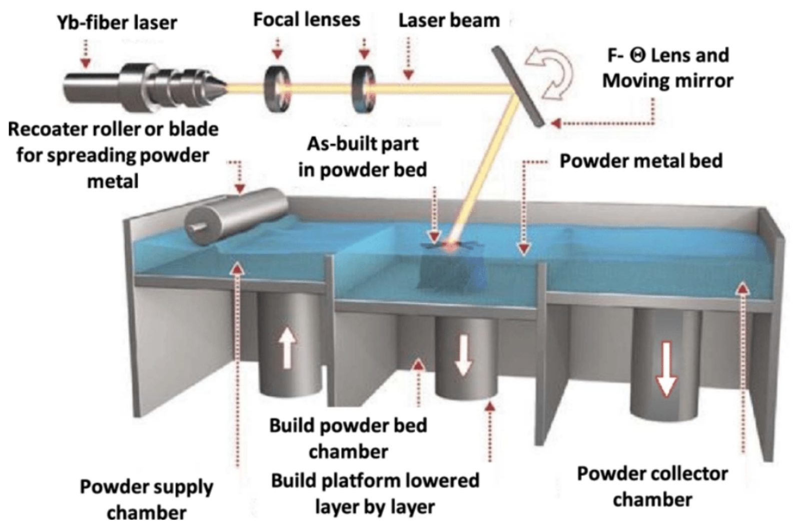

Yb-fiber laser
Focal lenses
Laser beam
F- Θ Lens and Moving mirror
Recoater roller or blade for spreading powder metal
As-built part in powder bed
Powder metal bed
Build powder bed chamber
Build platform lowered layer by layer
Powder supply chamber
Powder collector chamber

Figure 4.7: Working principle of powder-bed fusion is very similiar to the one of binder jetting. Source: [42]

Stereolitography

SLA (fig. 4.8) is one of the earliest if not the earliest method of 3D printing with the first prototype of SLA printer developed by the father of AM Charles Hull in 1986.

SLA is a vat polymerization method utilizing thin layer of a liquid precursor (resin or monomer solution) which after an exposure to UV light or an electron beam initiates a chain polymerization reaction which results in a solidification of exposed areas. This process also happens layer by layer until the 3D object is obtained. Despite the resin itself being able to provide some additional support to the object, support structures may also be needed in certain instances. Two most common methods are laser-SLA where the light is delivered in the form of a beam traversing a path on the layer, or digital light processing SLA where the whole layer is divided into pixels some of which (the ones of the desired shape on the layer) are then illuminated simultaneously.

Compared to other AM methods SLA can offer amazing precision, print quality, allowing for very precise and complex geometries to be printed, and in some cases even resolutions starting from less than 1 µm ([43]).

Despite most of the materials used show good mechanical properties, their range is very limited still ([44]). This combined with very slow printing and comparatively high costs makes the prospects of a wide spread of this technology rather small.

As mentioned above, SLA is very potent for manufacturing small and durable parts and is therefore often utilized in areas which can make use of this feature, such as biomedicine, chip industry but also aerospace and robotics.

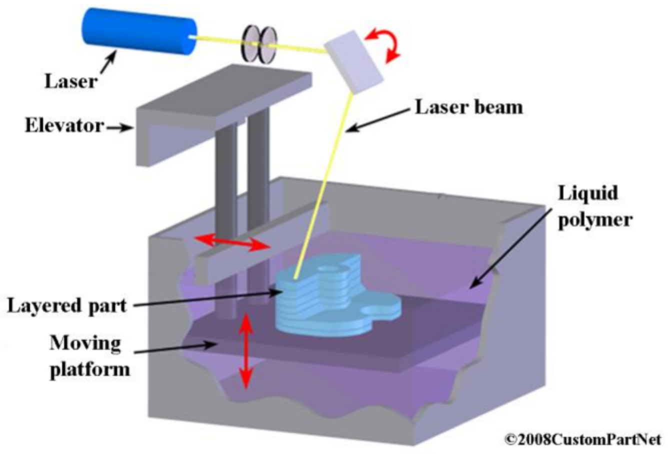


Figure 4.8: Working principle of stereolitography. Source: [45]

4.3 FDM – A closer look

4.3.1 On 3D printer itself

When it comes to FDM printers there are two common builds. **Cartesian** printers are more common. They have one or two motors for each x-, y- and z-axis, that is they can move in these three mutually perpendicular directions, hence the name. They usually have rectangular printing area. **Delta** printer on the other hand has three arms that come together at center and all three of them move at the same time at different rates to operate the nozzle. Delta printers also use Cartesian coordinates but their printing area tends to be circular.

In the following more detailed ”anatomy” of a 3D printer we will focus primarily on the Cartesian printers. In general most parts are similar for both the printers.

Software

3D printers require pre-processing of the models to be printed. This is usually mediated by slicing programs such as *Slic3r* and print control programs such as *pronterface*. These usually come bundled by the printer makers, most of which have their own dedicated programs.

Slicing program performs the tasks mentioned above, it orients the model, prepares eventual supports, slices the model and plans the printing paths while giving the user options for adjusting these parameters. The object is then saved into g-code which controls the process – that is not only the movement of the printer but also the temperature.

Example lines of a .gcode file:

```
M107
M104 S205 ; set temperature
G28 ; home all axes
G1 Z5 F5000 ; lift nozzle

; Filament gcode

M109 S205 ; set temperature and wait for it to be reached
G21 ; set units to millimeters
G90 ; use absolute coordinates
M82 ; use absolute distances for extrusion
G92 E0
G1 Z0.500 F7800.000 ; slowly move nozzle into position
G1 E-2.00000 F2400.00000
G92 E0
G1 X93.186 Y93.966 F7800.000
G1 E2.00000 F2400.00000
G1 F1305.89
G1 X94.382 Y92.870 E2.10244 ; slowly move nozzle into position
    (94.382,92.870)
G1 X95.576 Y92.011 E2.19533
G1 X97.679 Y90.986 E2.34307
G1 X99.436 Y90.715 E2.45532

  ...

G1 X100.541 Y100.240 E2.34140
G1 X100.422 Y100.362 E2.34540
G1 X100.072 Y100.282 F7800.000
G1 Z2.100 F7800.000
G1 E0.34540 F2400.00000
G92 E0
M107
; Filament-specific end gcode
;END gcode for filament

M104 S0 ; turn off temperature
G28 X0  ; home X axis
M84     ; disable motors
```

Hardware

1. The **Extruder** or printing head is a key component which deposits the filament. It consists of two sections: a cold end and a hot end.

   The cold end usually contains a motor moving a (also feed roller) which draws and pushes the filament into the extruder.

   The heat sink and heat sink fan are means to ensure the filament does not get melted prematurely before reaching the nozzle, while heater cartridge heats up the filament to the right temperature.

   All Extruder builds also contain a thermistor which checks and helps controlling the temperature.

Then finally the filament reaches the nozzle. Nozzle is where the level of detail is controlled. The most usual nozzle size has 0.4 mm diameter as turns out to be a fine compromise between the printing speed and quality.

Last part is Layer cooling fan which ensures the materials solidify properly. The slicer program usually determines whether the fan should be on or off based on the material.

2. **Printing bed** (also build platform) is the surface on which the final object is materialized. Typically, it consists of a sheet of glass, a heating element and surface that helps the filament stick but subsequently be easily removed after printing. Most printers come with all purpose surface, but best results are to be expected with dedicated surfaces for specific materials.

   As mentioned, some printing beds are heated in order to prevent the print-outs from warping while cooling. More on that in Filaments section.

   In recent years, most beds also come with some sort of self-calibrating system, that makes sure the bed is level with the nozzle. In case this system is not in place, calibrating has to be done by hand.

   In most Cartesian printers, the printing bed moves along the y-axis.

3. The **Motion control components** are parts responsible for movement of the printer along the three given axes. This is done by moving the printer bed (usually along the y-axis) and printer head (usually along x- and z-axis).

   Stepper motors are the keys to the mechanical movement of a printer. Unlike usual direct current (DC) motors, stepper motors rotate in increments or steps, which gives them better control over their position which is so much needed in 3D printing. During 3D printing, the printer moves all three motors simultaneously according to the gcode.

   Belts connected to motors, move the x- and the y-axis from side to side and are integral to the overall print speed and precision. The tension of the belts is important and most printers come with belt tensioning devices to keep them in optimal tightness.

   End stops are markers indicating the printers position along a given axis preventing it from moving past its print range. If a motor tries to move past this range, the end-stop will trigger causing the motor to stop.

These three are the core parts of a 3D printer. Naturally it contains more parts, those are not 3D printer specific like the ones mentioned above. These are components such as power supply, frame, motherboard, screens and user interfaces, stepper drivers and possibly SD card slots. A schematic 3D printer build can be seen in fig. 4.9

Filaments

This section presents some of the most common filaments used for FDM printing.

1. **Polyactic acid (PLA)** is a thermoplastic polyester usually obtained from fermented plant starch which makes it biodegradable. It is currently the most popular material for FDM printing for its leniency with printing conditions – it requires minimal supervision, has relatively low melting temperatures for printing (180 °C – 230 °C) and does not require heated bed.

   PLA does not excel at strength or durability and is not flexible.

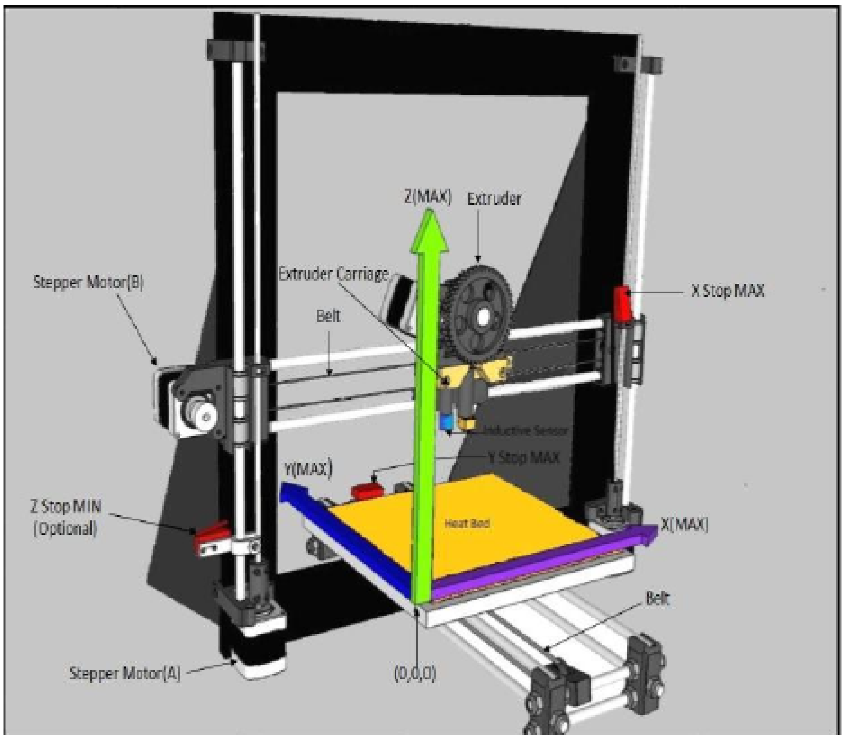


Figure 4.9: A schematic picture of 3D printer parts. Source: [34]

The advantages of PLA as a filament outweigh its mediocre mechanical properties and make it feasible for most consumer products.

2. **Acrylonitrile butadiene styrene (ABS)** is a thermoplastic polymer obtained by polymerizing styrene and acrylonitrile in the presence of polybutadiene, best known as the material of Lego bricks. It only recently got replaced as the most common material for 3D printing by PLA. ABS surpasses PLA in almost all mechanical aspects.

   ABS print temperatures are higher than the one of its predecessor ranging between 210 °C and 250 °C requires heated bed and often an enclosed printer as when it cools, it shrinks and when it cools too fast, it warps and deforms.

   For its mechanical properties ABS is good for printing various functional parts but the difficulties that come with printing cause its gradual decline in use.

3. **Polyethylene terephthalate glycol-modified (PETG)** is derived from the most common thermoplastic polymer resin – PET – by copolymerization, which lowers its melting temperatures and makes it more suitable for printing. It is a relatively new material for 3D printing, but already established itself as a good middle way between ABS and PLA. Whilst sharing many mechanical properties with and being more flexible than ABS, the printing itself is easier.

   PETG requires higher printing temperatures (about 220 °C – 250 °C) but does not require heated bed and does not warp like ABS.

   The spectrum of uses for PETG is very wide and keeps getting wider for it combines the main strengths of the other two most commonly used materials.

4. **Nylon** is a generic name for a family of thermoplastic synthetic polymers. Out of these nylon 610 and nylon 612 are the most commonly used for filaments.

   Nylon excels in all possible mechanical properties, it is very strong, flexible and durable.

   Printing using nylon requires high temperatures (240 °C – 260 °C) and a bed temperature ranging from 70 °C to 100 °C. It also suffers from bed adhesion issues, so it may require a layer of glue stick. Nylon also tends to absorb moisture easily which can cause a reduction in print quality, so it is better kept well sealed when not in use.

   Its great mechanical properties make it a multi-purpose material feasible for all kinds of products and industries. However the difficulties connected to printing using nylon prevent it from being used more widely.

5. **Thermoplastic polyurethane (TPU)** is a class of thermoplastic elastomers which are used for their specific properties different to those of the materials mentioned above including elasticity, flexibility, transparency and resistance to grease and abrasion. Printing temperatures of TPU are similar to the ones of PLA, working best at about 210 °C – 230 °C and with the printing bed preheated to 30 °C – 60 °C.

   Paradoxically the exact properties for which TPU is used tend to cause problems, therefore printing and the printing process requires supervision. This type of materials also has to be printed very slowly compared to other materials mentioned and is not very suitable for beginners.

There are many other materials suitable for this type of printing. In machines with multiple nozzles, dissolvable materials for support structures such as polyvinyl alcohol (PVA) and high-impact polystyrene (HIPS) can often be encountered.

4.4 3D printing in practice

4.4.1 Printing from Mathematica

Many present technical computing programs such as *Octave*, *Mathematica*, *Matlab* etc. support exports of their models in .stl format.

As an example we will make a model of a part of a hyperbolic paraboloid surface using *Mathematica* code. It is possible to print almost spacial data created by or imported to *Mathematica*. A good option is also using its vast libraries of 3D models. The only pitfall we have to watch for is that we have to make sure that the model has some thickness or is an enclosed object otherwise the export might produce a faulty code.

```
1 hyppar = RegionPlot3D[{x^2/8 - y^2/4.5 > z}, {x, -4, 4}, {y, -4,
    4}, {z, -1.5, 2.5},  BoxRatios -> Automatic]
2 Export["/home/Username/Directory/print/print.stl", hyppar , {"STL
    ", "BinaryFormat" -> False}]
```

Line 1 of the code creates the surface. First we designated its name. RegionPlot3D generates a part of a volume of a hyperbolic paraboloid. Intervals for each variable limit the surface to the desired part. BoxRatios set to Automatic maintains the proportions. The plot can be seen in figure 4.10

Line 2 uses Export function which takes our hyperbolic paraboloid and generates an .stl file of the displayed plot in file prints. The additional conditions are in place to assure the .stl file contains an ASCII code which we can then read rather then a binary format.

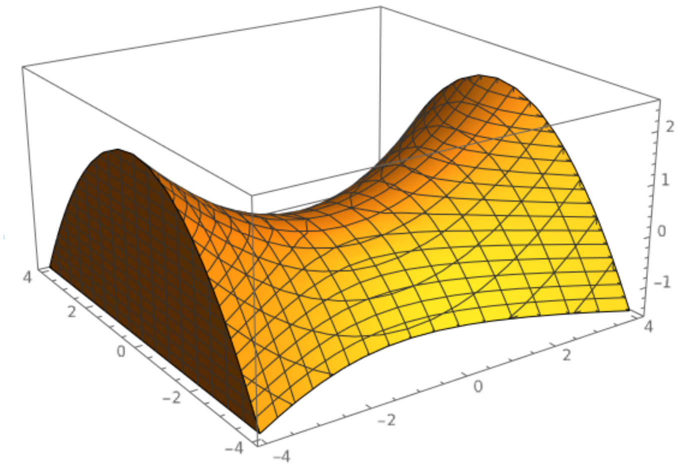


Figure 4.10: Hyperbolic paraboloid plot in Mathematica

With these settings the final model will have 8180 facets.

Example lines of the produced .stl file.

```
solid Created_by_the_Wolfram_Language_for_Students_-
   _Personal_Use_Only_:_www.wolfram.com
 facet normal 0 0 0
  outer loop
  vertex 2 1.604710817337036 -0.0714285746216774
  vertex 2.157087087631226 1.714285731315613 -0.0714285746216774
  vertex 2 1.714285731315613 -0.1474975049495697
  endloop
 endfacet

 facet normal 0 0 0
  outer loop
  vertex -2.157087087631226 1.714285731315613 -0.0714285746216774
  vertex -2.285714387893677 1.805524587631226 -0.0714285746216774
  vertex -2.285714387893677 1.714285731315613
   0.0001395089348079637
  endloop

  ...

 facet normal 0 0 0
  outer loop
  vertex -4 2.976353168487549 -0.6250348687171936
  vertex -4 3.290029287338257 -0.4062936007976532
  vertex -4 3.428571462631226 -0.6250348687171936
  endloop
 endfacet

endsolidCreated_by_the_Wolfram_Language_for_Students_-
   _Personal_Use_Only_:_www.wolfram.com
```

Notice the earlier mentioned feature of *Mathematica*. It does not compute facet normals of the surface or any model. This information is redundant for most slicing programs.

The final .stl file can now be opened in a slicing program (in our case *Slic3r*) and prepared for printing.

4.4.2 Printing from Blender

Also most contemporary modelling programs and CAD programs such as *Blender*, *Maya*, *SketchUp* and *Rhinoceros* naturally support .stl exports and are actually the main source of printable designs. Ways of obtaining a model in these programs were mentioned in earlier chapters. As an example print from *Blender* we will use one of its signature primitives – monkey Suzanne (fig 4.11).

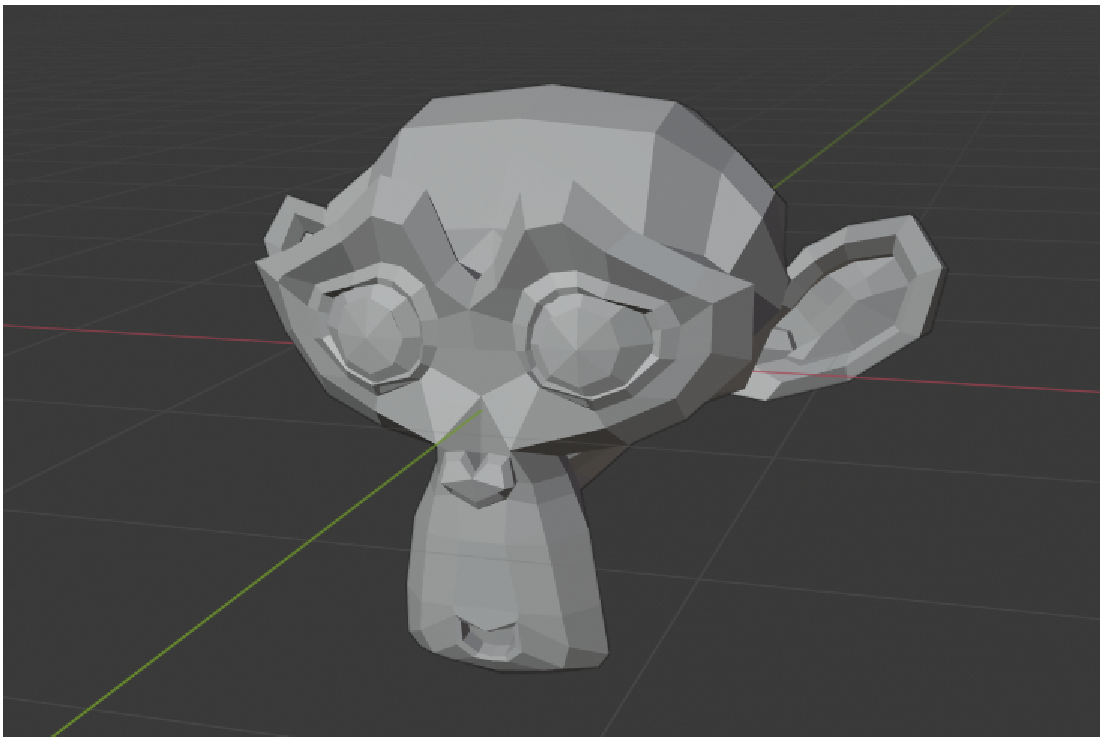


Figure 4.11: Monkey Suzanne.

After opening *Blender* we merely need to delete the original cube, then from the Add menu (Shift-A) select Mesh (M) and then Monkey (M). By doing this we are done with the sophisticated modelling part and can move on to export.

Make sure you have the Monkey model selected. In the File menu select Export (E) and then .stl (T) file type. After choosing a directory, in the region menu, it is worth checking some settings. If we want to make the file easily readable for us, it is worth checking the Ascii box. In our case we do not need to worry about other settings. If we had more models in one file, we can make sure we export only the ones needed by checking the Selection Only box. Then we can export.

The model should have 1002 facets.

Example lines of the produced .stl file:

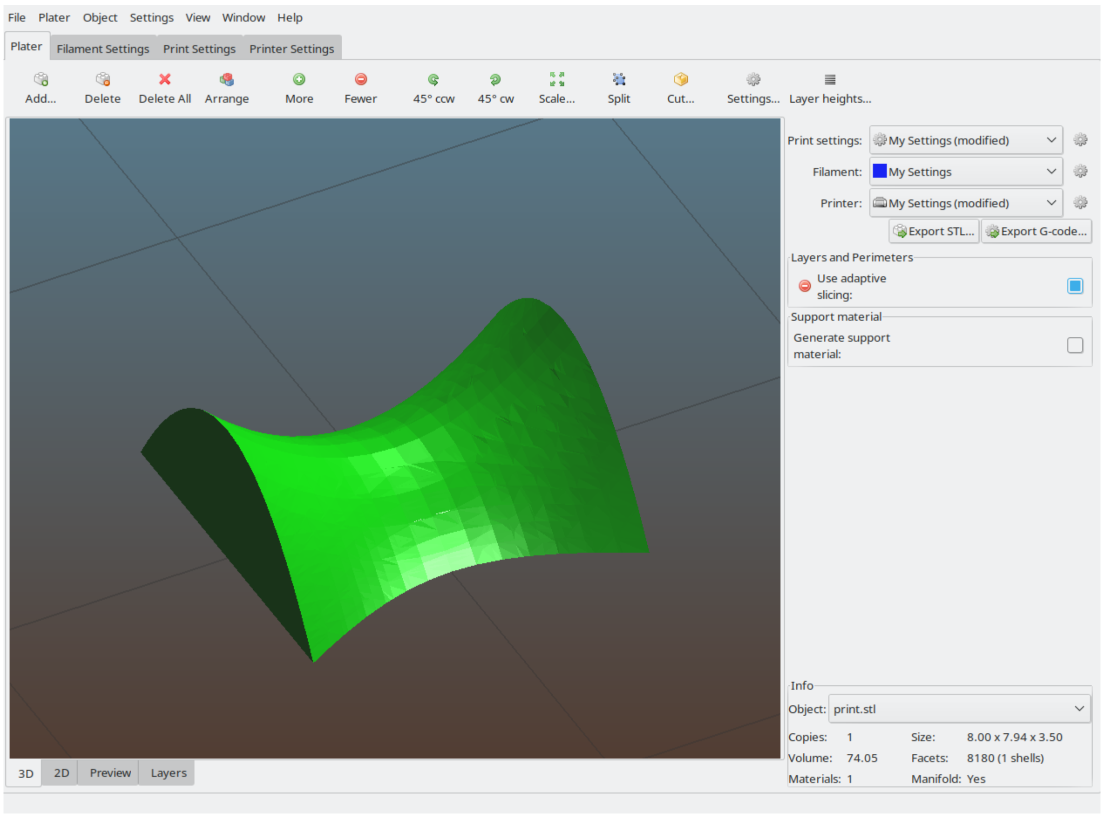
```
solid Exported from Blender-2.82 (sub 7)
facet normal 0.671345 -0.714459 -0.197092
outer loop
vertex 0.468750 -0.757812 0.242188
vertex 0.437500 -0.765625 0.164062
vertex 0.500000 -0.687500 0.093750
endloop
endfacet
facet normal 0.661707 -0.721862 -0.202628
outer loop
vertex 0.468750 -0.757812 0.242188
vertex 0.500000 -0.687500 0.093750
vertex 0.562500 -0.671875 0.242188
endloop

  ...

facet normal 0.068684 0.081767 -0.994282
outer loop
vertex -0.789062 0.328125 -0.125000
vertex -0.593750 0.164062 -0.125000
vertex -0.773438 0.125000 -0.140625
endloop
endfacet
endsolid Exported from Blender-2.82 (sub 7)
```

Note that unlike *Mathematica*, *Blender* does compute facet normals, and its vertex coordinates are way less precise. This big difference in precision reflects the need for bound tolerances mentioned in the section about slicing algorithms.

4.4.3 Slic3r

Slic3r is a free multipurpose slicing program useful for almost all printers. We already mentioned that most printers come with their own slicing software but the workflow in each is fairly similar. Slic3r is particularly on good terms with Prusa printers.

We open an .stl file by simply dragging it over the workspace. Newly added object can be manipulated but that often is not a good idea. The way it gets imported is usually sufficient (fig. 4.12).



Print settings: My Settings (modified)
Filament: My Settings
Printer: My Settings (modified)

Layers and Perimeters
Use adaptive slicing: ☑

Support material
Generate support material: ☐

Info
Object: print.stl
Copies: 1 Size: 8.00 x 7.94 x 3.50
Volume: 74.05 Facets: 8180 (1 shells)
Materials: 1 Manifold: Yes

Figure 4.12: Hypercolic paraboloid created in Mathematica in Slic3r environment.

The main interface consists of four main windows: Plater, Print settings, Filament settings and Printer settings. The last two pretty much speak for themselves and they take most of the information from a connected printer and/or is very dependent on the specific printer so consulting with printer manual is recommended.

Three important choices that have to be done by the user are: Print quality selection and filament selection.

Print quality selection (the main difference is layer height) is largely dependent on the issues discussed above, the higher quality, the longer an object takes to print but is smoother and more detailed. A case of low quality print settings is shown in fig. 4.13

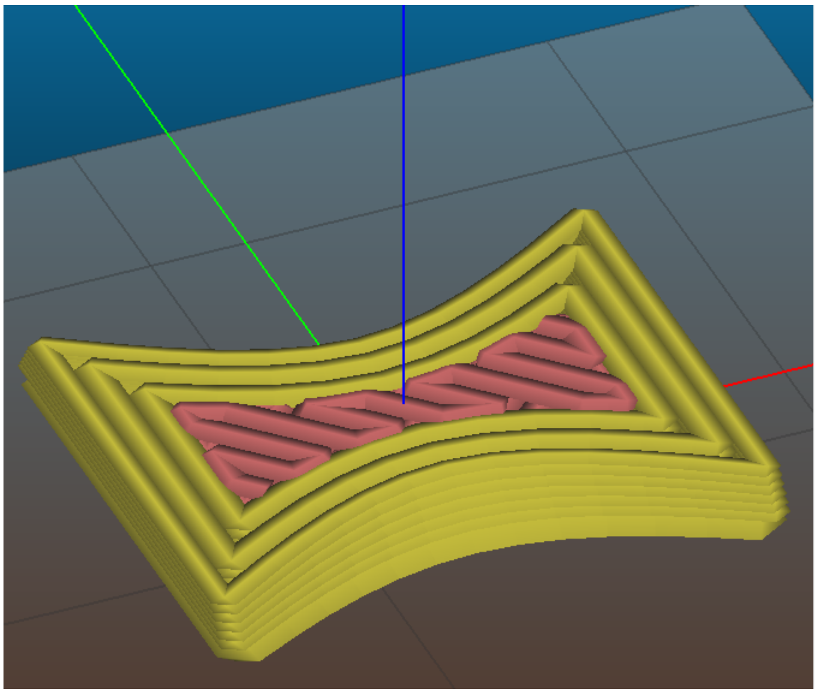


Figure 4.13: Finetuning printing quality is important.

Filaments largely depend on the intended use. PETG and PLA are by far the most common.

It is advised to check the slicing of an object in the Preview > Layers window before converting to gcode which is then sent to the printer for the final print (fig. 4.14).

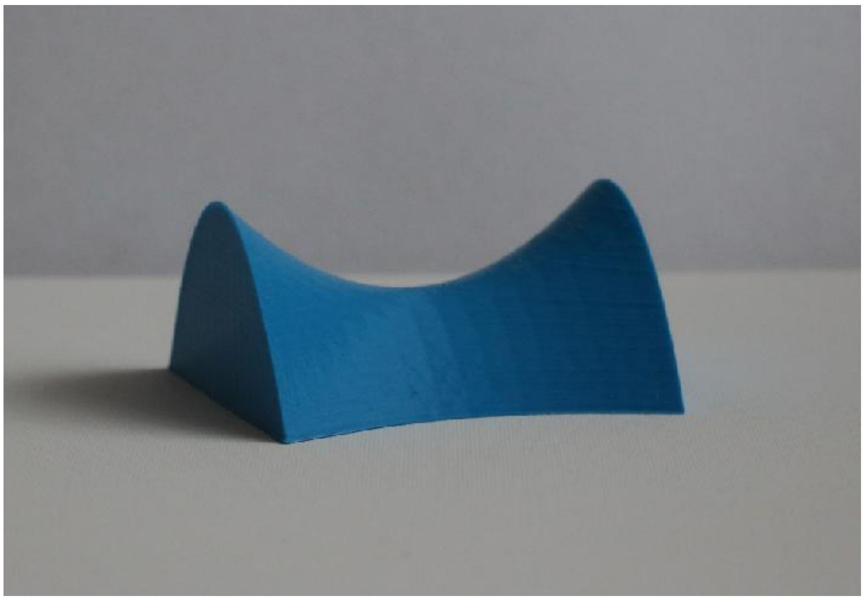


Figure 4.14: Modelled hyperbolic paraboloid printed in blue.

5. Augmented reality

In this chapter we will offer a brief look into the world of augmented reality (AR) which is an area of interest gaining a particular momentum in recent years. It draws a growing academic interest as well as the interest of many tech companies (fig. 5.1. What exactly is augmented reality though?

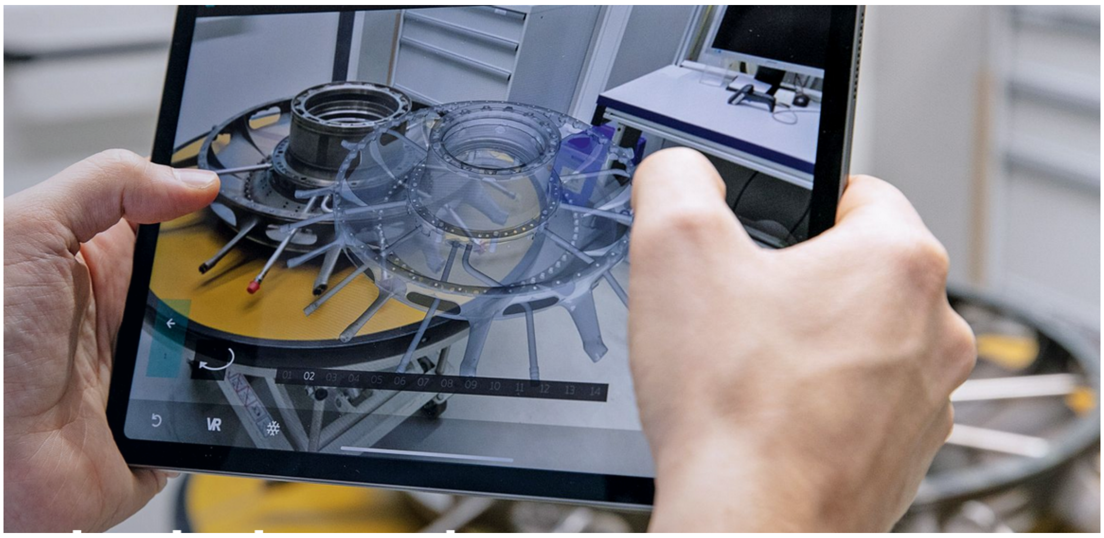


Figure 5.1: AR finds its way into various industries. Source: https://www.lufthansa-technik.com/augmented-reality

Answering this question is surprisingly hard and there is no consensus on a consistent definition. We will paraphrase one which is widely accepted amongst various articles on this topic, originating in one of the first comprehensive surveys on AR by Azuma ([46]): AR is a set of systems and technologies with three major characteristics: They

1. combine real and virtual,
2. are interactive in real time,
3. are registered in 3D.

The first condition excludes fully virtual environments such as virtual reality (VR). The interactivity condition excludes things such as 3D films, and the last one excludes objects such as 2D overlays – a camera app on a phone sees the real world and adds virtual controls, but these are not blended into the 3D scene.

Notice that this definition does not exclude augmenting a 3D scene by other things that objects, such as sounds.

The idea of AR is, that is enhancing the real world using virtual technology is as old as sci-fi and its first steps towards real life implementation go as far as to 1950s when Morton Heilig described the first idea of a possible realization of this idea while in 1968 the first head mounted display for augmented reality was made by Ivan Sutherland ([47]). Only recent advancements in computing technology and computer vision algorithms made this technology broadly viable so much so, that it is together with 3D printing and other computer aided technologies, sometimes called engines of the next industrial revolution. Is that not worth a deeper look?

This chapter will see much of its theoretical clutter alleviated from it as it uses many of the sets of rules (or similar ones) described in the lengthy chapter 3 ([48]). There we established several algorithms, the more optimized versions of are utilized by augmented reality for its navigation and orientation in space via a process called Simultaneous Localization and Mapping, or SLAM for short. This algorithm is able to simultaneously identify important features of a scene (the identified key features are way more sparse than in the case of SIFT[1] and less robust to noise), reconstruct its important points and find the position of a camera (sensor) within the scene, all this in real time (considering we have enough computational power). SLAM is very common in various areas of computer vision.

5.1 Reality, except better?

Augmented reality is but one of a bigger family of mixed realities on the Milgram's reality-virtuality spectrum (fig. 5.2).

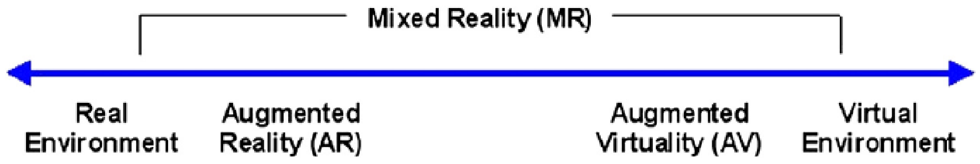


Figure 5.2: Milgram's reality-virtuality spectrum. Source: [50]

Virtual reality

Although virtual reality is not technically a mixed reality, it definitely deserves its couple of paragraphs as it undoubtedly is a very prominent technology with close ties to AR.

Unlike the other two, virtual reality environment does not have to live time process any other input than the one from a user, who is completely immersed in a virtual environment. Virtual environments may or may not be based on actual places but exist apart from current physical reality. The input usually comes in a form of a controller or via motion control system, which are the only instances when computer vision plays its role.

Virtual reality usually operates with way bigger data sets than AR, especially when it tries to achieve high realism. For VR it is necessary to recreate the entire environment, that is, all models, textures, lighting and so on, while AR merely adds singular objects to a scene even though it has to go through some sort of understanding and structuring the world in which it operates, but that is rather a problem of data processing than data storing.

Though, most known for its use in experience and gaming oriented industries, it is gaining momentum in areas such as art and design – both industrial and graphical (5.3).

Augmented virtuality

Augmented virtuality (AV) represents an inverse of augmented reality of sorts. It is a primarily virtual environment, not necessarily immersive like in the case of VR, which

[1]In AR, SIFT is usually replaced by FAST (Features from Accelerated Segment Test)[49].

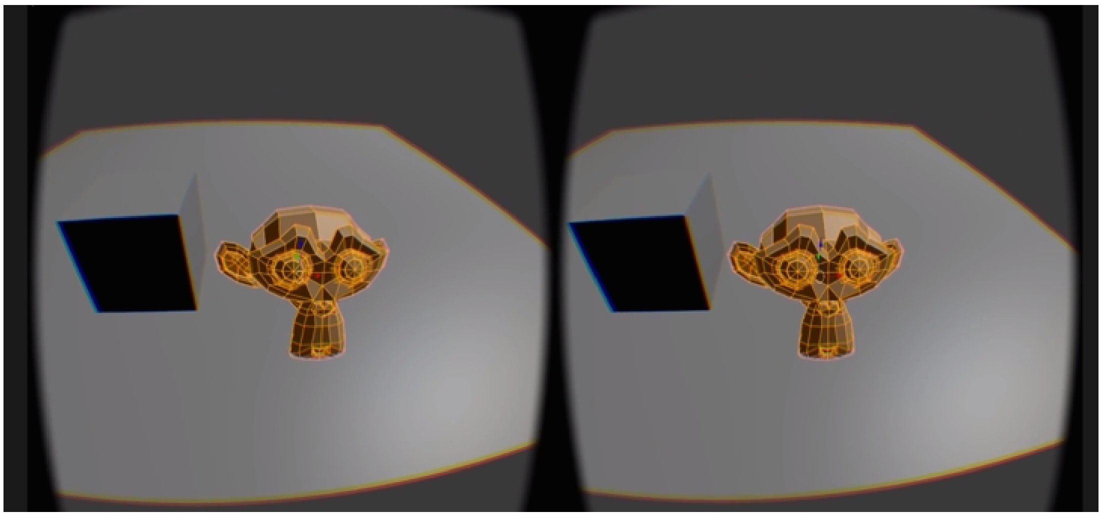


Figure 5.3: Blender VR viewport. Source: `https://www.blendernation.com/2015/04/24/`

is enhanced by imported real life objects. These can be things such as projections of ones hands (fig. 5.4) for better manipulation with the virtual environment or videos playing in predefined frames.

A good example of such application in use is Google's StreetLearn, which is a DeepMind backed desktop app implemented within Google StreetView offering augmented virtuality environment ([52]).

With progressing technology, the difference between AR and AV becomes more and more blurry and we might witness them merge into one soon ([52]).

5.1.1 Augmented reality – a walk through shallow waters

As, mentioned, will not dive too deep into the waters of AR. In this subsection we will offer a short overview of some devices, shortcomings and applications of AR with references for those who would like to stroll a bit further from the beach.

Displays

There are several ways of sorting displays that can be used for the purposes of AR, we adopt one from ([47]). There exist three major types:

1. Head mounted displays (HMD),
2. handheld displays,
3. spatial displays.

HMD is a device that preceded all the others in the form of I. Sutherland's patent from 1968 and comes in a form of a display device worn on the users head in the form of glasses, helmets, headsets or recently even contact lenses (`https://www.mojo.vision/mojo-lens`). They are mostly designed to be see-through, that is they leave the user's view unobstructed while adding a graphical overlay through the means of mirrors, transparent displays or miniature projectors. Obstructed view devices less suitable for HMDs but are also used. Obstructed view device does not let the user see right through it and rather feeds them with information through a display transmitting a camera recording of the surroundings. Though many attempts of mass-marketing AR

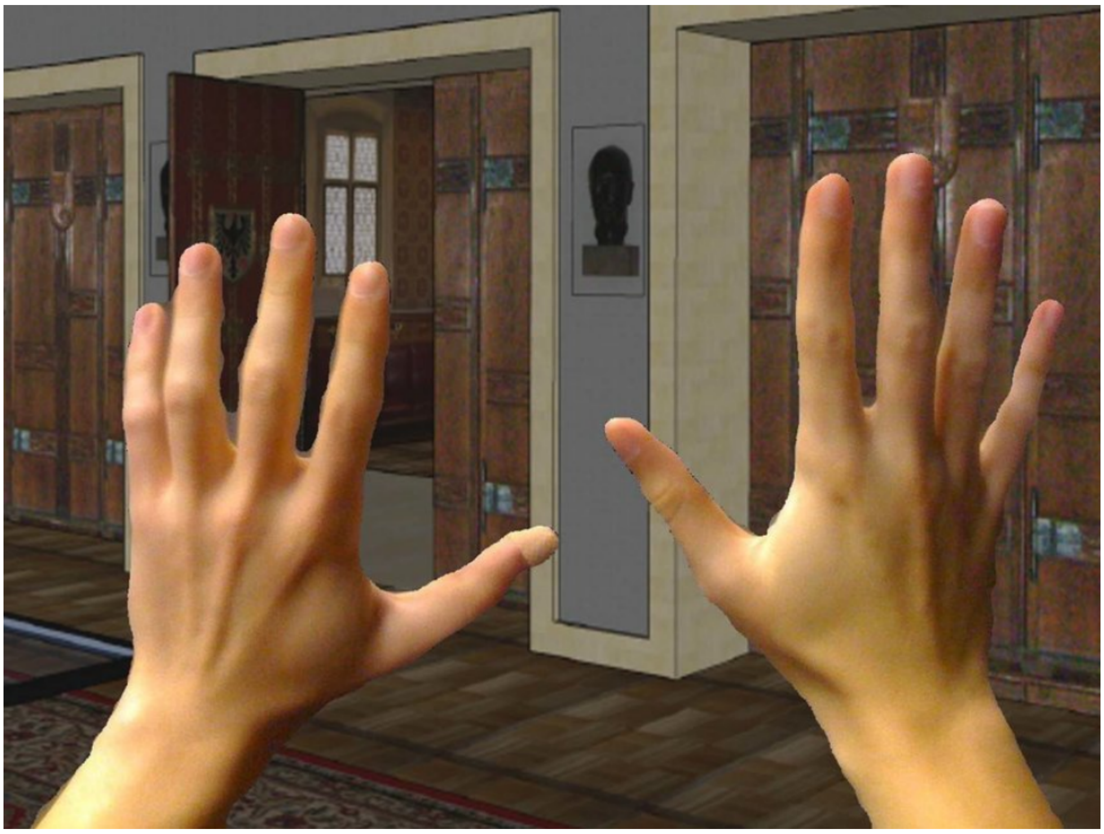


Figure 5.4: Virtual environment, real hands. Source: [51]

glasses were made, they never became a wide spread technology, unlike most handheld devices.

Handheld devices hold a prominent position amongst all the AR display methods as they are the most widely spread devices capable of AR simulation. Devices such as tablets, mobile phones, but surprisingly even laptops and PCs fit into this category. They are mostly, although not exclusively, view obstructed. They do not have to deal with fashion problems (like AR glasses) and can usually offer high quality cameras and great computational power in form of strong CPUs and GPUs. Amongst these devices, when it comes to AR, the brightest future probably belongs to tablets as they combine the strength of PCs and laptops, that is large screens, with portability of mobile phones.

Spacial displays produce a special type of AR called spacial augmented reality (SAR). SAR makes use of video projectors, optical elements, holograms and other tracking technologies in order to display graphics directly into the physical environment without the need for an actual display. This expands the amount of users that can interact with the virtual model at the same time which is naturally an attractive feature for schools, companies, scientific labs, museums and art communities. These systems are, more than the other two, still in need of more development. More general information about them can be found for instance in [53] and a model use case in [54].

In the following subsection we will focus mainly on handheld mobile devices.

Pipeline in a nutshell

As in the case of photogrammetry, even this process starts with image acquisition, this time in the form of a video. Video is a little more than a big set of images which can be processed. As in the case of photogrammetry, a 3D reconstruction of a scene would not be possible from a single frame, here video has a small edge since it collects new data with even the slightest camera movement. However, if the scene is to be understood and errors minimized it is best to 'wiggle' the camera around.

SIFT, as it was described in chapter 3, is not very suitable for this kind of image reconstruction. The process through which key features are extracted in SIFT are very computationally expensive and lengthy. It is beneficial for photogrammetry as it can easily deal with noise and finds a large amount of very precise matches amongst various images, which highly increases the precision of a reconstruction.

For the purposes of AR precision and robustness is partially traded for speed. Currently unrivalled method in this area is Features accelerated segment test (FAST). Unlike SIFT that falls in the family of blob detection methods, FAST is a corner detection method focused mainly on differences in pixel colours. More details can be found in [49].

FAST is the first step within the frame of Simultaneous Localization and Mapping (SLAM) ([48]). Unlike structure from motion, though based on similar premises, SLAM does not aim to reconstruct a 3D model of the scene perceived by the camera, that would again be too computationally expensive. SLAM only aims to find its location within the space at a given time and understand the spacial relation between reconstructed points, which it remembers. 'At a given time' is a short phrase suggesting that contrary to SfM, SLAM also works with ordered linear data and since it often happens that two consequential camera positions are rotated by only a small angle, its triangulations tend to suffer from a slightly higher margins of error.

The sparse point cloud created through this process turned out to be an interesting article for a number of tech companies and many apps utilizing AR store the acquired data on clouds. This can be a double edged sword, on one hand enhancing the AR experience of a user using the app on the same location, on the other hand such data mining can be exploited and is not only morally questionable.

Much of the modern software for AR purposes is equipped for recognizing some simple properties of the scene as well, such as planar surfaces, some basic bodies, text and equipped with physics engine and proper tools for depth estimation treat them as such when a model is placed within a scene.

Generally the AR pipeline looks as shown in fig. 5.5, from image acquisition it takes two separate paths, one for displaying real scene and one for the virtual part.

Placing a model into a scene and letting it exist there relative only to the computed points does not yield good results. As everything is computed on a fly, the model tends not to have a stable position relative to the real world and looks and behaves unnatural within the scene's context.

For these reasons special markers called *anchors* are used as a link between the virtual object and the real world. An anchor carries its own virtual coordinate system in which the virtual part of the AR can be represented. The position of an anchor compared to the world coordinates is absolute and adapts its updates in each frame. The anchor then updates objects attached to it accordingly.

There are many types of anchors and differ across various platforms. Probably the most common anchor is an image. Today very commonly used image type anchors are QR codes (fig. 5.6), but almost any image, given it is distinctly recognizable within the space (such as logos, text, distinct patterns,...), can be used as an anchor. The software might get confused if the anchor image can be recognized in the scene several times, therefore simple patterns such as uniformly coloured rectangles are not suitable.

There are other types of anchors, such as 3D objects (those usually have to be photoscanned before they are suitable for anchors, faces or possibly planes within the scene if a software can easily detect and distinguish them.

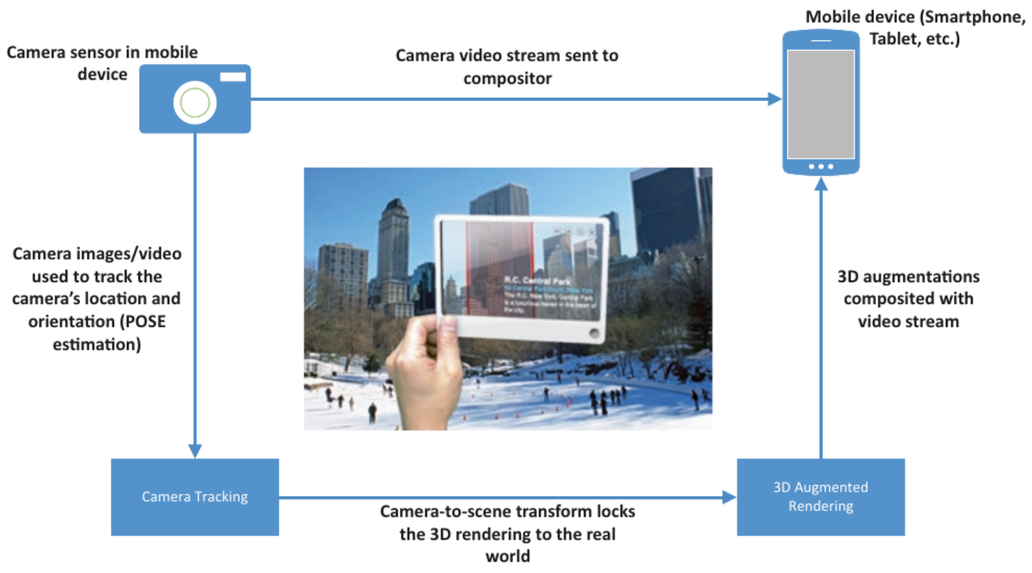

Camera sensor in mobile device
Camera video stream sent to compositor
Mobile device (Smartphone, Tablet, etc.)
Camera images/video used to track the camera's location and orientation (POSE estimation)
3D augmentations composited with video stream
Camera Tracking
Camera-to-scene transform locks the 3D rendering to the real world
3D Augmented Rendering

Figure 5.5: AR pipeline. Source: [55]

A special type of anchor is a geographic or location anchor. This anchor uses GPS data as its localization. This type of anchor was utilized by one of the most widely spread AR apps at the time Pokémon GO (fig. 5.7). Geographic anchor is easy to use across multiple devices without the need for each of the devices to be able to reconstruct the scene features.

Added 3D objects are modelled as meshes in the same fashion as in a virtual scene.

Lives augmented

The world of applications for AR is basically limitless and probably still beyond our imagination today. As the computational power of all devices will keep increasing, algorithms for AR improving and numbers of developers and everyday users increasing, AR is bound to become a part of everyday life.

Even today, elements of AR and VR are integrated in CAD and CAM software, making it possible to interact with models on a completely new level, making them accessible for a wider scale of users, who can become well trained professionals for fields such as architecture, engineering and construction.

Many companies already experiment with large scale world scans (Scape Technologies) for AR overlays opening the door to commercial and enterprise uses. Looking outside through a window through a camera lens in a flat in Prague will be able to show you companies, shops or restaurants in the surroundings or a view from a window in Rome.

A huge beneficiary of further AR development might be especially education. AR can allow experience and interaction with hypothetical terms and situations on a completely new scale. If we believe the findings of E. Dale, we remember only about 10 percent of what we read (yes, of all the pages of this thesis only up to 10 percent will stick, that is why it is so long, so you learn something, you know), but about 50 percent of what we hear and see and up to 90 percent of what we create and experience first hand. AR can open the door to a new paradigm of education where students will be able to see, create and experience. Wide and far would also reach the options for not only learning by also specialized training, be it medical training, military training or training of pilots.

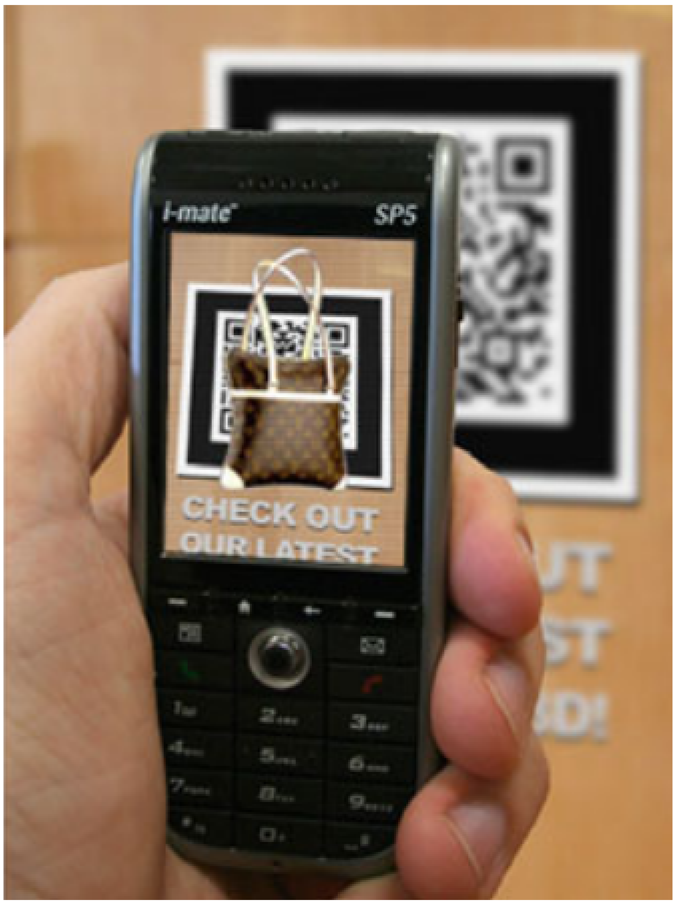


Figure 5.6: QR codes serve as a very good anchor for AR. Source: [56]

Often debated is utilizing AR in medicine for fast diagnoses, assistance with surgical procedures, compensation for patients with disabilities and many more.

A comprehensive account of recent developments in AR, its applications, workings and much more can be found in a digestible form in [55] and [47]. A good overview but with a deeper look in AR development and technical background are in [57] and [58].

5.1.2 A simple path to an AR experience

DIY augmented reality became, well, reality in recent years. One of the free software allowing to build an AR is Unity. Unfortunately, technical limitations prevented the author from presenting the pipeline of creating an AR app. We can atleast recommend some reading: `https://unity.com/learn` .

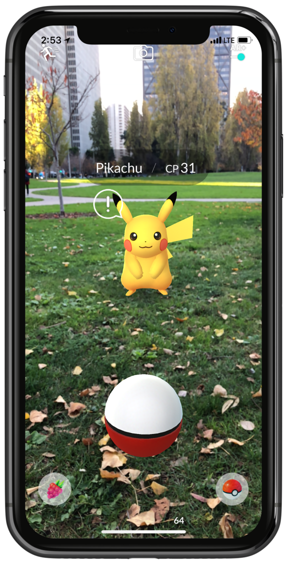


Figure 5.7: Pokémon GO is a popular AR game using GPS anchors. Source: `https://pokemongolive.com/post/arplus`

6. Algorithm

In this chapter we will employ the knowledge obtained in some of the previous chapters in order to create a simplified program representing NURBS surfaces. The programming language of choice is Python for its easy readability and suitability for graphics.

The main aim here is to show how some of the algorithms described above can be used and implemented specifically. Therefore, we will cut some corners and use some of the plentiful packages of Python for things unrelated to the algorithms themselves and will not pay much attention to the miscellaneous parts of the code which again are not directly linked to the functionality of said algorithms. Full code producing a virtual environment for viewing pre-specified NURBS (fig. 6.1) can be found in attachments.

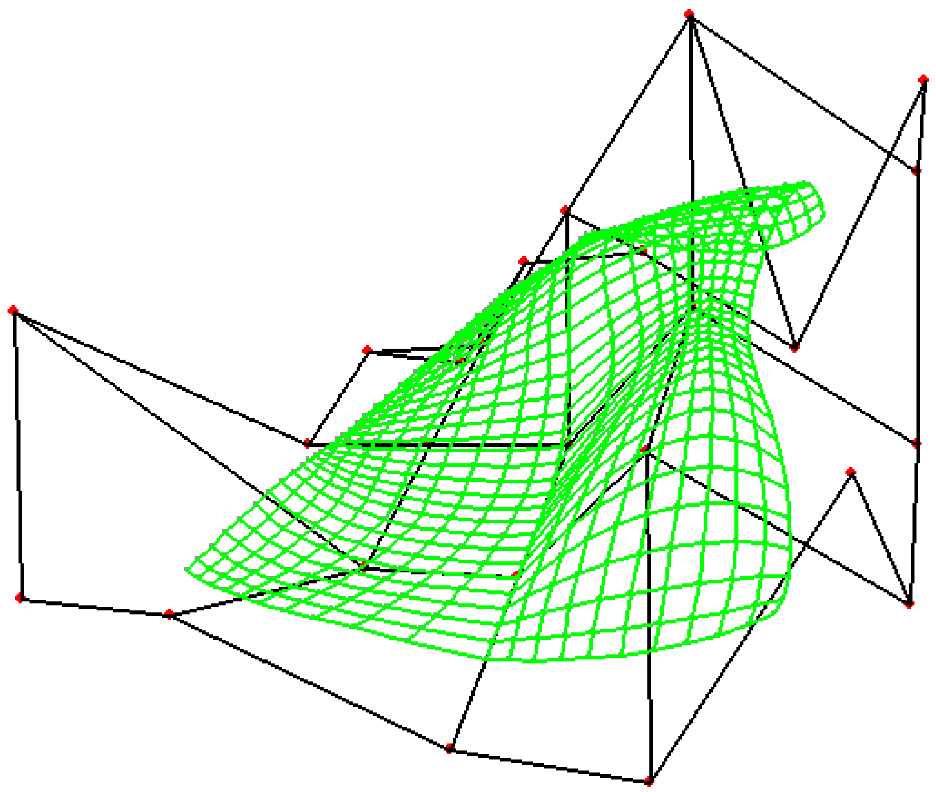


Figure 6.1: NURBS surface represented by contour lines in the parametric directions. Notably visibility is not solved within the program and whatever was created last gets drawn atop other objects.

6.1 Points of a NURBS surface

We mentioned that when displaying, analytical surfaces are interpolated. This part will focus on finding points for the said interpolation. We want to keep things at their basic level, so we will only display the surface using isolines.

Commonly used parts of the code

We omit the generic ones such as `for`, `if`, or self explanatory such as `in range` etc.

- The `len()` function returns the number of items in an object.
- `list=[]` – empty square brackets declare that an object is an empty list
- `list.append()` – adds an item at the end of a list
- `my_div(a,b)` divides `a` by `b` and returns `0` if `b=0` instead of `NaN`.
- `list[i]` accesses the i-th item of a list
- `list=[[item] for item in list]` switches all items of a list to lists
- `np.arrange(a,b,s)` consequentially returns all values between `a` and `b` with step `s`
- `int()` returns integer value of an object if possible

User input

A user can insert the values they please in control_points.py:

```python
verts = [[[-2, 2, 1], [-1, 2, 1], [0, 2, 0], [1, 2, 0], [2, 2, 2]],
         [[-2, 1, 2], [-1, 1, 0], [0, 1, 0], [1, 1, 1], [2, 1, 0]],
         [[-2, 0, 0], [-1, 0, 0], [0, 0, 0], [1, 0, 1], [2, 0, 0]],
         [[-2, -1, 0], [-1, -1, 0], [0, -1, 1], [1, -1, 2], [2, -1, 1]],
         [[-2, -2, -1], [-1, -2, 0], [0, -2, 0], [1, -2, -1], [2, -2, 1]]] #insert point coordinates


def control_vertices():
    net=[item for elem in verts for item in elem]
    met = []
    for i in range(len(net)):
        met.append([[x] for x in net[i]])
    return met


num_pts_u = len(verts)
num_pts_v = len(verts[0])
```

and in knot_vectors.py:

```python
weights = [[1, 3, 1, 1, 1], [1, 1, 1, 1 / 2, 1], [4, 1 / 2, 1, 4, 1], [1, 1, 1, 1, 1], [2, 3, 1, 2, 1]]  # enter weigths
knotvec_u = [0, 1, 2, 3, 4, 5, 6, 7]  # enter knot vector in the u-direction of length from num_pts_u+3 to 2*num_pts_u
knotvec_v = [-3, -2, 0, 1, 1, 2, 3, 4]  # enter knot vector in the v-direction of length from num_pts_v+3 to 2*num_pts_v
step = 0.1 #determines how dense the izolines will be, the lower the value, the denser the net
           # (very low values slow the program significantly(depends on the range of values of knot vectors))
```

6.1.1 Basis functions

Input: Number of points in the u-direction, knot vector u, value u

Output: Value of basis functions $N_{i,k}$ at the point u

```
 1 def basis_fc(u, knotvec_u, num_pts_u):
 2     n_i = []
 3     a = len(knotvec_u) - num_pts_u
 4     if a == 0:
 5         for i in range(len(knotvec_u) - 1):
 6             if knotvec_u[i] <= u < knotvec_u[i + 1]:
 7                 n_i.append(1)
 8             else:
 9                 n_i.append(0)
10         n_i = [[x] for x in n_i]
11         return n_i
12     elif a > 0:
13         if u == knotvec_u[-1]:
14             for i in range(len(knotvec_u) - 1):
15                 if knotvec_u[i] < u <= knotvec_u[i + 1]:
16                     n_i.append(1)
17                 else:
18                     n_i.append(0)
19         else:
20             for i in range(len(knotvec_u) - 1):
21                 if knotvec_u[i] <= u < knotvec_u[i + 1]:
22                     n_i.append(1)
23                 else:
24                     n_i.append(0)
25         for k in range(1, a):
26             n_i_1 = []
27             for j in range(len(n_i) - 1):
28                 n_i_1.append(
29                     my_div((u - knotvec_u[j]) * n_i[j], knotvec_u
    [j + k] - knotvec_u[j]) +
30                     my_div((knotvec_u[j + k + 1] - u) * n_i[j +
    1], knotvec_u[j + k + 1] - knotvec_u[j + 1]))
31             n_i = n_i_1
32         n_i = [[x] for x in n_i]
33         return n_i
34     else:
35         return '0'
```

For sake of simplification, we do not give the user the freedom to choose the order of the basis function, we compute is as length of the knot vector - number of points in a direction. If this is 0, only the first step of basis function $N_{i,0}$ computation is executed and we are left with something like $[0, 1, 0, 0, 0, 0]$.

For $a > 0$, we precompute this first step on lines 20–24 and use them for computing higher order basis functions (25–3), which we then retrieve as a list of lists such as $[[0.31231], [0.32532], [0], [0]]$.

6.1.2 Points of a NURBS surface

Input: Coordinate matrix, basis functions, weights, step for interpolation

Output: Coordinates of NURBS surface points for interpolation.

```
 1 def Q_coord(m):
 2     A_x = point_coord(points, m, num_pts_u, num_pts_v)
 3     Q_x = []
 4     Q_x_i = []
 5     m = len(knotvecc_u) - num_pts_u
 6     n = len(knotvecc_v) - num_pts_v
 7     for v in np.arange(knotvecc_v[n - 1], knotvecc_v[len(
   knotvecc_v) - n] + 0.0001, step):
 8         r_k_v = basis_fc(v, knotvecc_v, len(A_x[0]))
 9         for u in np.arange(knotvecc_u[m - 1], knotvecc_u[len(
   knotvecc_u) - m] + 0.0001, step):
10             r_k_u = basis_fc(u, knotvecc_u, len(A_x))
11             Q_x_1 = multiplyMatrix(multiplyMatrix(np.transpose(
   r_k_u), compMatrix(weights, A_x)), r_k_v)
12             Q_x_2 = multiplyMatrix(multiplyMatrix(np.transpose(
   r_k_u), weights), r_k_v)
13             Q_x_i.append(my_div(Q_x_1[0][0], Q_x_2[0][0]))
14     Q_x.append(Q_x_i)
15     return Q_x
```

Line 2 helps us extract the m-th point coordinate individually. In the for cycles on lines 7–12 we compute the numerator (Q_x_1) and denominator (Q_x_2) of the fraction in NURBS definition with a step entered by the user for all viable knot spans. We then divide these numbers and retrieve a list of values for the m-th coordinate of a point.

```
def nurbs_pts():
    nurbs_pts = []
    Q_x = Q_coord(0)[0]
    Q_y = Q_coord(1)[0]
    Q_z = Q_coord(2)[0]
    for i in range(len(Q_x)):
        nurbs_pts.append([[Q_x[i]], [Q_y[i]], [Q_z[i]]])
    nurbs_ptts = []
    [nurbs_ptts.append(x) for x in nurbs_pts]
    if [[0],[0],[0]] in nurbs_ptts:
        nurbs_ptts.remove([[0],[0],[0]])
    return nurbs_ptts
```

This part uses the function above to glue the individual point coordinates together.

6.2 Transforming surface

This part (main.py) contains a lot of code for displaying the surface using pygame package. It creates a window of $595 \times (\sqrt{2} \cdot 595)$ pixels. It allows us to define drawing functions and accepts user input from keyboard or mouse. We will only look specifically on the part we can recognize from reading above.

Input: 3D coordinates of NURBS points

Output: NURBS points drawn on the screen with isolines in chosen view

```
    rotationX = [[1, 0, 0], [0, math.cos(RotX), -math.sin(RotX)],
    [0, math.sin(RotX), math.cos(RotX)]]
    rotationY = [[math.cos(RotY), 0, -math.sin(RotY)], [0, 1, 0],
    [math.sin(RotY), 0, math.cos(RotY)]]
    rotationZ = [[math.cos(RotZ), -math.sin(RotZ), 0], [math.sin(
  RotZ), math.cos(RotZ), 0], [0, 0, 1]]
```

Defines rotation matrices. `RotX, RotY, RotZ` are initially set to 0 but can be manipulated with numerical keyboard (2 and 8 rotate about the x axis, 4 and 6 about the y axis and 1 and 3 about the z axis)

```
    indexx: int = 0
    projectedPoints2 = [p for p in range(len(nurbs))]
    for point in nurbs:
        rotated = multiplyMatrix(rotationX, point)
        rotated = multiplyMatrix(rotationY, rotated)
        rotated = multiplyMatrix(rotationZ, rotated)

        z = 1 / (distance - rotated[2][0])
        projection = [[z, 0, 0], [0, z, 0]]

        projected = multiplyMatrix(projection, rotated)

        x = int(projected[0][0] * size) + objPos[0]
        y = int(projected[1][0] * size) + objPos[1]

        projectedPoints2[indexx] = [x, y]
        pygame.draw.circle(display, grey, (x, y), 1)
        indexx += 1
```

Transforms the points to screen coordinates. First rotations are applied still in space. `distance` is again the user input, originally set to 5. It is the distance between the object and camera in space. x- and y-values (positions in the viewing window) are consequentially computed and points drawn.

```
    for n in range(num_u-1):
        for nn in range(num_v):
            ConnectPoint2(n +nn*num_u , (n+1) % num_u + nn *
   num_u, projectedPoints2)

    for m in range(num_u):
        for i in range(num_v - 1):
            ConnectPoint2(m + i * num_u, m + num_u * (i + 1),
   projectedPoints2)le(display, grey, (x, y), 1)
        indexx += 1
```

Draws isolines as a linear interpolation between the neighbouring points. `ConnectPoint2(i,j,p)` is a customized function drawing green line segments between i-th and j-th point of list p.

Starting view is shown in fig. 6.2:

Influence of a negative weight can be seen in fig. 6.3. The influence of high weight is shown in fig. 6.4.

The program did not go through debugging. Some issues may occur, for instance when zooming too close. It is kept at a basic level and can be further greatly improved with time.

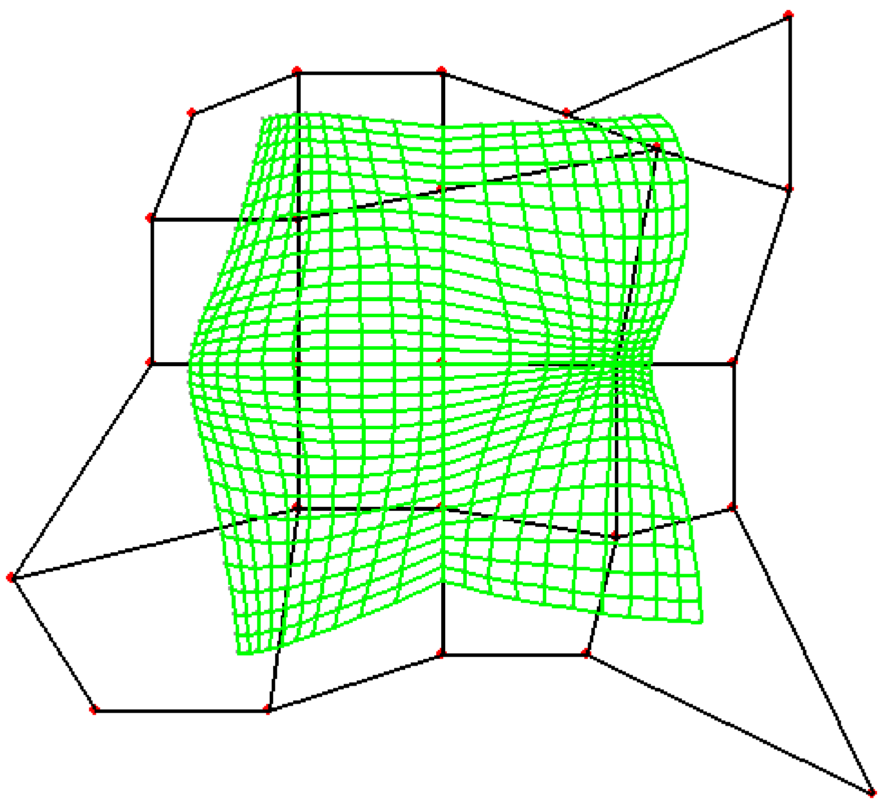


Figure 6.2: The view of the surface can be manipulated via keyboard.

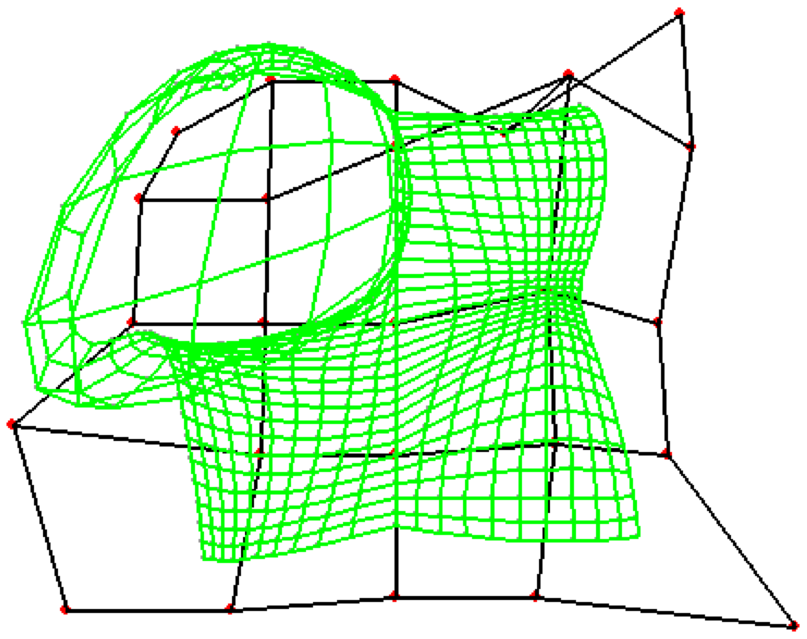


Figure 6.3: Negative values correctly cause surface distortions.

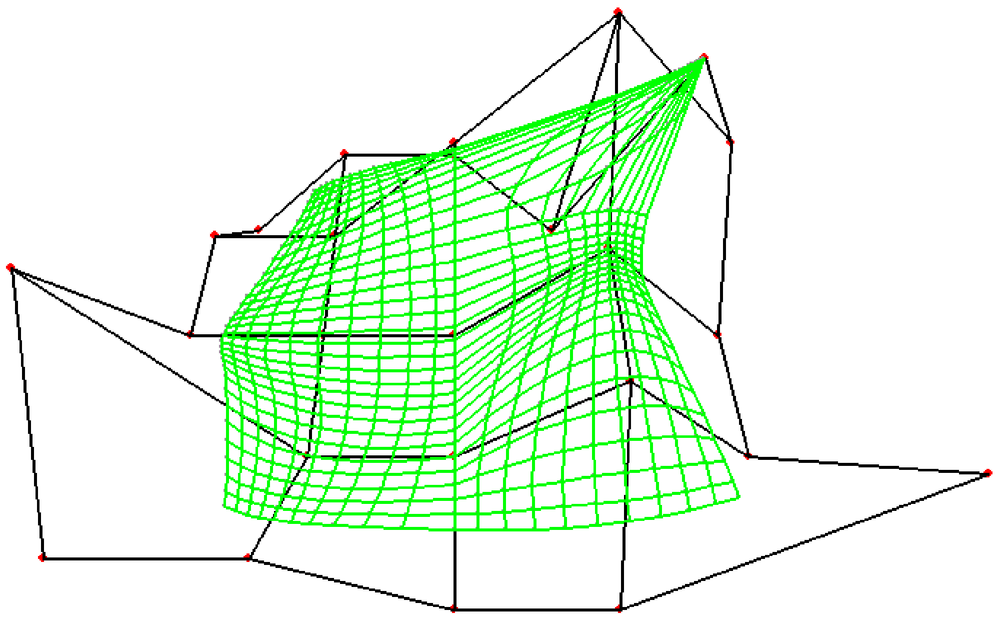


Figure 6.4: High weights draw surface towards the point with high weight.

Conclusion

In this thesis we offered a peek into the world of 3D computer aided modelling and its applications. We were offered a look into the workings of a modelling program, photogrammetric software, 3D printing and augmented reality.

This work was in its nature a survey trying to bring casual users closer to 3D graphics and mathematics, perhaps help them solve potential problems they encountered in their modelling process, and show students of mathematics where and how their knowledge can be applied.

When studying these topics I was appalled by its vastness and depth but also reasonable accessibility for newcomers with a decent mathematical background and no programming skills. I tried to pass on my positive impression of this field in hope to draw more people into this world of endless possibilities.

Bibliography

[1] Peter Shirley. *Fundamentals of computer graphics.* AK Peters, Wellesley, Mass, 2005.

[2] Manfredo Carmo. *Differential geometry of curves and surfaces.* Prentice-Hall, Englewood Cliffs, N.J, 1976.

[3] Nicholas M. Patrikalakis Takashi Maekawa. *Shape Interrogation for Computer Aided Design and Manufacturing.* Springer Berlin Heidelberg, February 2002.

[4] David F. Rogers. *An Introduction to Nurbs: With Historical Perspective.* MORGAN KAUFMANN PUBL INC, July 2000.

[5] J. Žára, B. Beneš, J. Sochor, and P. Felkel. *Moderní počítačová grafika.* Computerpress, Brno, 2004.

[6] Eliška Chudáčková. Computer representation of surfaces. Master's thesis, Faculty of Mathematics and Physics, Charles Univesity, 2017.

[7] G. Farin, J. Hoschek, and M. S. Kim. *Handbook of Computer Aided Geometric Design.* ELSEVIER SCIENCE and TECHNOLOGY, 2002.

[8] Ning Wang. Analysis on the innovation to sculpting with the aid of 3d digital technology. 01 2015.

[9] Victor J. Duvanenko, Ronald S. Gyurcsik, and W.E. Robbins. Simple and efficient 2d and 3d span clipping algorithms. *Computers and Graphics*, 17(1):39–54, 1993.

[10] Fuhua Cheng, Fengtao Fan, Conglin Huang, Jiaxi Wang, Shuhua Lai, and Kenjiro T. Miura. Chapter 5: Smooth surface reconstruction using doo-sabin subdivision surfaces. In *Proceedings of the 2008 3rd International Conference on Geometric Modeling and Imaging*, GMAI '08, page 27–33, USA, 2008. IEEE Computer Society.

[11] Helmut Pottmann. *Architektural geometry.* SpringerBentley Inst. Press, WienNew York, NYExton, Pa, 2010.

[12] Sherif Ghali. *Introduction to Geometric Computing.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[13] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–, 11 2004.

[14] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In Aleš Leonardis, Horst Bischof, and Axel Pinz, editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[15] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. *ACM Trans. Graph.*, 25(3), 2006.

[16] Onur Özyeşil, Vladislav Voroninski, Ronen Basri, and Amit Singer. A survey of structure from motion. *Acta Numerica*, 26:305–364, may 2017.

[17] David Nistér and Henrik Stewénius. Scalable recognition with a vocabulary tree. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:2161 – 2168, 02 2006.

[18] Andrew Zisserman Richard Hartley. *Multiple View Geom Comp Vision 2ed*. Cambridge University Press, February 2019.

[19] Duane Brown. Close-range camera calibration. *Photogramm. Eng.*, 37, 12 2002.

[20] Theo Moons, Luc Van Gool, and Maarten Vergauwen. *3D Reconstruction from Multiple Images, Part 1*. Now Publishers Inc, October 2009.

[21] Michael Parker. Digital signal processing 101 (second edition). In *Digital Signal Processing 101 (Second Edition)*. Newnes, second edition edition, 2017.

[22] Riccardo Gherardi, Michela Farenzena, and Andrea Fusiello. Improving the efficiency of hierarchical structure-and-motion. pages 1594–1600, 06 2010.

[23] Siyu Zhu, Runze Zhang, Lei Zhou, Tianwei Shen, Tian Fang, Ping Tan, and Long Quan. Very large-scale global sfm by distributed motion averaging. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[24] Johannes L. Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4104–4113, 2016.

[25] Manolis Lourakis. A brief description of the levenberg-marquardt algorithm implemened by levmar. *A Brief Description of the Levenberg-Marquardt Algorithm Implemented by Levmar*, 4, 01 2005.

[26] Heiko Hirschmuller. Stereo processing by semi-global matching and mutual information. *in IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:328–341, 02 2008.

[27] Frédéric Cazals and Joachim Giesen. Delaunay triangulation based surface reconstruction: Ideas and algorithms. Technical report, INRIA, 2004.

[28] Nina Amenta. The crust algorithm for 3d surface reconstruction. In *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, SCG '99, page 423–424, New York, NY, USA, 1999. Association for Computing Machinery.

[29] Michal Jancosek and Tomas Pajdla. Exploiting visibility information in surface reconstruction to preserve weakly supported surfaces. *International Scholarly Research Notices*, 2014:1–20, 08 2014.

[30] Baoyun Guo, Jiawen Wang, Xiaobin Jiang, Cailin Li, Benya Su, Zhiting Cui, Yankun Sun, and ChangLei Yang. A 3d surface reconstruction method for large-scale point cloud data. *Mathematical Problems in Engineering*, 2020:1–14, 08 2020.

[31] Wen Wang, Tianyun Su, Haixing Liu, Xinfang Li, Zhen Jia, Lin Zhou, Zhuanling Song, and Ming Ding. Surface reconstruction from unoriented point clouds by a new triangle selection strategy. *Computers and Graphics*, 84:144–159, 2019.

[32] Hao Chen and Jie Shen. Denoising of point cloud data for computer-aided design, engineering, and manufacturing. *Engineering with Computers*, 34, 07 2018.

[33] Edward Aboufadel, Sylvanna V. Krawczyk, and M. Sherman-Bennett. 3d printing for math professors and their students. *ArXiv*, abs/1308.3420, 2013.

[34] Jyoti Sekhar Banerjee and Arpita Chakraborty. Analysis of implementation factors of 3d printer: The key enabling technology for making prototypes of the engineering design and manufacturing. *International Journal of Computer Applications*, pages 8–14, 03 2017.

[35] Athanasios Anastasiou, Charalambos Tsirmpas, Alexandros Rompas, Kostas Giokas, and Dimitris Koutsouris. 3d printing: Basic concepts mathematics and technologies. pages 1–4, 11 2013.

[36] William Oropallo and Les A. Piegl. Ten challenges in 3d printing. *Engineering with Computers*, 32(1):135–148, jun 2015.

[37] Donghong Ding, Zengxi Pan, Dominic Cuiuri, Huijun Li, and Stephen van Duin. Advanced design for additive manufacturing: 3d slicing and 2d path planning. In *New Trends in 3D Printing*. InTech, jul 2016.

[38] S.H. Choi and K.T. Kwok. A tolerant slicing algorithm for layered manufacturing. *Rapid Prototyping Journal*, 8:161–179, 08 2002.

[39] Jingchao Jiang and Yongsheng Ma. Path planning strategies to optimize accuracy, quality, build time and material use in additive manufacturing: A review. *Micromachines*, 11:633, 06 2020.

[40] G.Q. Jin, W. Li, C.F. Tsai, and Lihui Wang. Adaptive tool-path generation of rapid prototyping for complex product models. *Journal of Manufacturing Systems - J MANUF SYST*, 30:154–164, 08 2011.

[41] Mohsen Ziaee and Nathan Crane. Binder jetting: A review of process, materials, and methods. *Additive Manufacturing*, 28, 2019.

[42] Tuğrul Özel, Ayça Altay, Bilgin Kaftanoğlu, Nicola Senin, Richard Leach, and M. Donmez. Focus variation measurement and prediction of surface texture parameters using machine learning in laser powder bed fusion. *Journal of Manufacturing Science and Engineering*, 142:011008, 11 2019.

[43] Christina Schmidleithner and Deepak M. Kalaskar. Stereolithography. In Dragan Cvetković, editor, *3D Printing*, chapter 1. IntechOpen, Rijeka, 2018.

[44] Tuan D. Ngo, Alireza Kashani, Gabriele Imbalzano, Kate T.Q. Nguyen, and David Hui. Additive manufacturing (3d printing): A review of materials, methods, applications and challenges. *Composites Part B: Engineering*, 143:172–196, 2018.

[45] Ivan Hanzlicek and M. Pentek. Computational modeling of stereolithography. In *IEEE GSC*, 2014.

[46] Ronald T. Azuma. A Survey of Augmented Reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 08 1997.

[47] *Handbook of Augmented Reality*. Springer-Verlag GmbH, November 2011.

[48] Gerhard Reitmayr, Tobias Langlotz, Daniel Wagner, Alessandro Mulloni, Gerhard Schall, Dieter Schmalstieg, and Qi Pan. Simultaneous localization and mapping for augmented reality (pdf). *International Symposium on Ubiquitous Virtual Reality*, 0:5–8, 07 2010.

[49] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. volume 3951, 07 2006.

[50] Alan B. Craig. *Understanding Augmented Reality: Concepts and Applications.* MORGAN KAUFMANN PUBL INC, June 2013.

[51] Gerd Bruder, Frank Steinicke, Kai Rothaus, and Klaus Hinrichs. Enhancing presence in head-mounted display environments by visual body feedback using head-mounted cameras. pages 43–50, 01 2009.

[52] Stefaan Ternier, Roland Klemke, Marco Kalz, Patricia Ulzen, and Marcus Specht. Ar learn: Augmented reality meets augmented virtuality. *Journal of Cheminformatics - J Cheminf*, 18, 01 2012.

[53] Ramesh Raskar Oliver Bimber. *Spatial Augmented Reality: Merging Real and Virtual Worlds.* A K PETERS LTD (MA), August 2005.

[54] S. Casas, J. Gimeno, Pablo Casanova-Salas, José V. Riera, and C. Portalés. Virtual and augmented reality for the visualization of summarized information in smart cities: A use case for the city of dubai. 2020.

[55] Jon Peddie. *Augmented Reality.* Springer-Verlag GmbH, April 2017.

[56] Dieter Schmalstieg, Tobias Langlotz, and Mark Billinghurst. Augmented reality 2.0. pages 13–37, 01 2008.

[57] Erin Pangilinan, Steve Lukas, and Vasanth Mohan. *Creating Augmented and Virtual Realities.* O'Reilly Media, Inc, USA, April 2019.

[58] Paul Mealy. *Virtual and Augmented Reality For Dummies.* John Wiley and Sons Inc, July 2018.

List of Figures

A. Attachments

A.1 Python code

Full code, the parts of which were shown in chapter 6 is included in the attached .zip file: For full functionality all files (control_vertices.py, knot_vectors.py, main.py, matrix.py, nurbs.py) have to be used in parallel. Alternatively, the code can be found on `https://mega.nz/folder/HXpUURTY#I5GfJYbStgW54DKWfbJuGw`.