I would like to thank my supervisor, RNDr. Josef Pelikán, for his time and valuable insights during my work on the thesis. Also, I would like to thank the government of the Czech Republic for awarding me the scholarship which made my studies in Prague possible and changed my life for the better. Finally, a big thank you to my parents for showing me that life is not a straight line and there’s always time for change.

I declare that I have written this bachelor thesis on my own and entirely using the cited sources. I agree to lending of this thesis and its publishing.

In Prague, May 28th, 2010

Ivan Jankovič
## Contents

1 Introduction .......................................................... 6  
   1.1 Viewpoint quality ............................................... 6  
   1.2 Project goals .................................................. 7  

2 Methods for viewpoint selection ................................. 8  
   2.1 Historical background ......................................... 8  
   2.2 The algorithm for finding the best viewpoint ............... 9  
   2.3 Heuristic measure ............................................. 11  
   2.4 Weighted heuristic measure ................................... 11  
   2.5 Viewpoint entropy ............................................ 12  

3 Implementation ....................................................... 14  
   3.1 Tools and resources ........................................... 14  
   3.2 Using OpenSceneGraph ......................................... 15  
   3.3 Implementing the algorithm ................................... 16  
   3.4 Internal structure of the program ............................ 20  

4 Results ........................................................................ 24  

5 Conclusion .................................................................... 29  
   5.1 Future work ....................................................... 30  

Bibliography ................................................................... 31  

A CD contents .............................................................. 33  

B User guide .................................................................. 34  
   B.1 System requirements ............................................... 34  
   B.2 Usage ................................................................. 34
List of Figures

1.1 A 3D scene viewed from two different viewpoints . . . . . . . . . . . . . . . . . . . 6
2.1 Objects as building blocks of the scene . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Polygons as building blocks of objects . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1 A real-world 3D scene . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 Pablo Picasso’s “Head of a woman” . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.3 Scene processing in precomputation phase . . . . . . . . . . . . . . . . . . . . . . 18
3.4 Creating scene viewpoints . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.5 Refining the search for the best viewpoint . . . . . . . . . . . . . . . . . . . . . . 20
4.1 Discretization of a 3D truck model . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.2 Heuristic measure vs. Viewpoint entropy . . . . . . . . . . . . . . . . . . . . . . . 26
4.3 Weighted heuristic measure with different weight proportions 26
4.4 Bias towards the number of faces in WHM . . . . . . . . . . . . . . . . . . . . . . 27

Klíčová slova: Nejlepší pohled na 3D scénu; Entropie pozorovacího místa; Automatické umístění kamery; Zkoumání virtuálních světů.

Title: Best view of 3D scene
Author: Ivan Jankovič
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor’s e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: The computation of good viewpoints is useful in many fields: exploration of virtual worlds, global illumination, robot motion, radiosity, image-based modeling, etc. The main goal of the thesis was to implement several methods for computing the optimal viewpoint of a 3D scene. Two methods have been implemented - Heuristic Measure and Viewpoint Entropy. In addition, a modification of the first method, Weighted Heuristic Measure, was created and tested to find the values that give the best results. For the purpose of method evaluation, a simple application was developed, which loads a 3D scene and runs the implemented methods.

Keywords: Best view of 3D scene; Viewpoint entropy; Automatic camera positioning; Exploration of virtual worlds;
Chapter 1

Introduction

1.1 Viewpoint quality

The computation of good viewpoints is useful in many fields: exploration of virtual worlds, visual servoing, global illumination, robot motion, radiosity, image-based modeling, etc. as stated in [1]. However, there is no widespread consensus about the meaning of the term “good view” in computer graphics. By using common sense, we can derive that viewpoint quality is in direct proportion to how much information about a scene we get from that viewpoint. For instance, if we compare the two views of a given scene in Figure 1.1, we could say that the second view is better, because it gives us more insight into the placement of objects in the scene.

Figure 1.1: A 3D scene viewed from two different viewpoints
Our conclusion seems straightforward, but if we were to delegate this decision making to a computer, we would find that converting our subconscious thought processes into a series of well-defined steps of an algorithm is not necessarily trivial. Many different approaches in tackling this particular problem have been taken over the years, some of which will be mentioned in the next chapter.

1.2 Project goals

The main goals of this thesis are:

- to implement several methods for computing the best viewpoint of a 3D scene
- to create a simple application for examining the performances of the implemented methods
- to discuss the possible improvements to those methods.

In the next chapter, we mention some of the earlier approaches to the problem of finding the best viewpoint of a scene, we present the algorithmic concept of our approach and introduce the methods for computing the quality of an individual viewpoint which were implemented as a part of this project. Chapter 3 explains some of the design choices made during the development process, describes the graphical library used in the program and elaborates on the internal structure of the application. Chapter 4 presents the outputs of the program, some interesting facts that were discovered about the implemented methods, as well as some aspects of the algorithm structure that could be approached differently. Chapter 5 concludes with an overview of the work done in the project and lists some areas of the program that could be improved in the future. Appendix A provides a list of contents on the accompanying CD, and Appendix B contains a short guide for starting the program and its manipulation.
Chapter 2

Methods for viewpoint selection

In this section, we will cover briefly the different approaches to the problem of computing good viewpoints, and present in more detail the methods that have been implemented as a part of this thesis.

2.1 Historical background

In the last fifteen years, computation of good viewpoints has become a widely researched topic. As its field of application grew, so did the number of different methods and approaches to making it more precise and efficient. Many of those approaches, however, were being based on certain assumptions, which were afterwards shown not to hold for all cases. For example (examples taken from [1]), Kamada and Kawai considered a viewing direction to be good if it minimized the number of degenerated faces under orthographic projection. This method failed when comparing scenes with equal number of degenerated faces, and it could not guarantee that the user would see a large amount of details. An improvement of that approach, by Barral et al, had problems with objects containing holes, as these were not captured properly by the algorithm.

In dealing with creation of image-based virtual reality, Bourque and Dudek [3] looked at viewpoint quality from a different perspective. When creating the environment, they searched for the most interesting views, where interest was measured by the extent to which a location attracts visual attention. For that purpose, they used a theory from human psychophysics,
which suggests that various types of geometric structure such as line endings, oriented line segments, or curves “pop out” of an image when they are different from the rest of the scene.

Arbel and Ferrie [4] considered the development of an active vision system for moving through a familiar environment with the task of identifying and localizing known objects. This system was based on so-called entropy maps - object probabilities associated with image measures, which were used for determining the ambiguity of given camera positions. By choosing viewpoints that minimize ambiguity, the system seeks out locations that are maximally informative.

2.2 The algorithm for finding the best viewpoint

In order to find the best viewpoint of a 3D scene, we first need to examine the structure of that scene and find out what kind of information it contains. Next we need to establish a well-defined method for determining the quality of any given viewpoint, deriving only from the information available to us. That means not only using the original information, but also any additional precomputed data that we are able to build into the scene. Finally, having a set of viewpoints, we are able to find the one with the highest quality according to our method.

In general, a 3D scene consists of one or more geometrical objects, such as buildings, vehicles, living creatures, etc. For example, the scene in Figure 2.1 contains several buildings. Those buildings are comprised of walls, doors, windows, and so on, until we reach the basic building blocks of each object. One way of thinking about objects in a scene is as groups of polygons put together. This is presented in Figure 2.2, where every polygon is filled with a different color. A polygon is a concept which a computer understands, and can deal with in a standardized fashion. The computer does not know if a group of polygons that it stores represents an aeroplane or a kitchen table. This is why the algorithm for finding the best viewpoint of a 3D scene must deal only with the concept of polygons and the information associated with them.

As stated in [1], there are two parameters that are especially related to the quality of a viewpoint: the projected area of the objects and the number of faces seen. These parameters only give good results when combined; when
Figure 2.1: Objects as building blocks of the scene

Figure 2.2: Polygons as building blocks of objects
just the first parameter is used, we have no insight into the amount of detail seen; using only the second parameter can give bad results with objects containing areas with a high number of small polygons. This is why it is desirable to have a function that takes both parameters into account and balances them well.

In the rest of this chapter, we present the methods implemented as a part of this thesis, we cover briefly their mathematical background and describe the way they use the two important parameters mentioned above to get the best results. The actual implementation of the methods is not discussed until the next chapter.

2.3 Heuristic measure

As stated in [2]: “The quality of a viewpoint of a scene can be computed by the following heuristic measure:

\[
C(V) = \frac{\sum_{i=1}^{n} \left\lceil \frac{P_i(V)}{P_i(V) + 1} \right\rceil}{n} + \frac{\sum_{i=1}^{n} P_i(V)}{r} \tag{2.1}
\]

where \( V \) is the viewpoint, \( C(V) \) is the viewpoint quality of the scene or object, \( P_i(V) \) is the number of pixels corresponding to the polygon \( i \) in the image obtained from the viewpoint \( V \), \( r \) is the total number of pixels of the image (resolution of the image), and \( n \) is the total number of polygons of the scene. In this formula, \( \left\lceil x \right\rceil \) denotes the smallest integer greater than or equal to \( x \). The first term in (2.1) gives the fraction of visible surfaces with respect to the total number of surfaces, while the second term is the ratio between the projected area of the scene (or object) and the screen area (thus, its value is 1 for a closed scene”). Both terms can have values between 0 and 1, and they have the same impact on the final result.

2.4 Weighted heuristic measure

While the previous method established the important factors of a scene and used them in a “1-1” proportion for finding the best viewpoint, we thought it would be interesting to experiment with that proportion and compare the results. The weighted heuristic measure takes the two terms in (2.1)
and multiplies them by two different constants (weights), so that one term contributes more to the final result than the other. If we denote the two constants as $W_1$ and $W_2$, we have

$$C(V) = W_1 \frac{\sum_{i=1}^{n} \left[ \frac{P_i(V)}{P_i(V) + 1} \right]}{n} + W_2 \frac{\sum_{i=1}^{n} P_i(V)}{r}$$

(2.2)

We can also think of the non-weighted heuristic measure as a special case of weighted heuristic measure, where $W_1$ and $W_2$ are both equal to 1.

### 2.5 Viewpoint entropy

As stated in [2]: “The Shannon entropy of a discrete random variable $X$ with values in the set $\chi = \{x_1, x_2, ..., x_n\}$ is defined as

$$H(X) = -\sum_{i=1}^{n} p_i \log p_i$$

(2.3)

where $n = |\chi|$ and $p_i = Pr[X = x_i]$ for $i \in \{1, ..., n\}$. The entropy gives us the average information or uncertainty of a random variable. For continuity, we use the convention that $0 \cdot \log 0 = 0$.

To define viewpoint entropy, the relative area of the projected faces over the sphere $S$ of directions centered in the viewpoint $V$ is used as probability distribution. Thus, the viewpoint entropy is defined by

$$H(V) = -\sum_{i=0}^{N_f} \frac{a_i}{a_t} \log \frac{a_i}{a_t}$$

(2.4)

where $N_f$ is the number of faces of the scene, $a_i$ is the projected area of face $i$ over the sphere, $a_0$ represents the projected area of background in open scenes, and $a_t = \sum_{i=0}^{N_f} a_i$ is the total area of the sphere. In a closed scene, or if the viewpoint does not see the background, the whole sphere is covered by the projected faces and consequently $a_0 = 0$. Hence, $\frac{a_i}{a_t}$ represents the visibility of face $i$ with respect to viewpoint $V$. The maximum entropy is obtained when a certain viewpoint can see all the faces with the same projected area $a_i$. So, in an open scene, the maximum viewpoint entropy is $\log(N_f + 1)$ and, in a closed scene, it is equal to $\log N_f$. The best viewpoint
is defined as the one that has maximum entropy, i.e., maximum information captured.

The main drawback of viewpoint entropy is that it depends on the polygonal discretization. A highly discretized region will heavily attract the attention of the measure. Also, in the way it is defined, this measure is dependent on the background. Nevertheless, the background can be considered or not into the calculations, depending on whether we want to emphasize or not the relative area covered by the projection.”

In our program, we will work exclusively with open scenes, which is given by the algorithm used for placing the camera in the scene - we take a bounding sphere of the complete scene and compute the camera positions on it, which means that the camera can never be placed inside of an object. The algorithm for camera positioning will be described in more detail in the next chapter.
Chapter 3

Implementation

3.1 Tools and resources

The main goal of this thesis was to implement several methods for finding the best viewpoint of a 3D scene, as a part of a simple experimental application which could demonstrate the results of running those methods. One of the requirements was that the program be able to work with common 3D file formats, and view them using GPU acceleration. To accomplish this, several choices had to be made, including choosing the programming language, graphics API and target operating system. The program also had to provide for extending the list of implemented methods, which required a modular design, and therefore using the object-oriented paradigm was self-imposing.

In our case, the choices were made from the top. After examining many different comparisons between OpenGL and Direct3D (such as [5], [6]), we decided to use OpenGL, as it fully sufficed for our needs, and it had much less overhead than its opponent. Since there was no intention nor logic in building everything from scratch using low-level graphics calls, a good graphics toolkit was needed. After some searching and analysis, we opted for OpenSceneGraph, an open source high performance 3D graphics toolkit, used by application developers in fields such as visual simulation, games, virtual reality, scientific visualization and modeling [7]. We will present some of the toolkit’s features in the next section.

When choosing the programming language for the project, we considered the standards in the area of computer graphics, as well as personal experience and preferences. The initial set included C++, Java and C, in order of preference. Since only the first two languages supported OpenSceneGraph, and
the latter did so through so-called Community projects [8] (non-professional individuals programming the OpenSceneGraph bindings for Java), we picked C++ as the language of choice.

3.2 Using OpenSceneGraph

OpenSceneGraph is a 3D graphics library based around the concept of a scene graph, providing an object-oriented framework on top of OpenGL. This frees the developer from implementing and optimizing low-level graphics calls and provides many additional utilities for rapid development of graphics applications [8].

A scene graph is a hierarchical tree data structure that organizes spatial data for efficient rendering [9]. As stated in [10]: “A node may have many children but often only a single parent, with the effect of a parent applied to all its child nodes; an operation performed on a group automatically propagates its effect to all of its members. In many programs, associating a geometrical transformation matrix at each group level and concatenating such matrices together is an efficient and natural way to process such operations. A common feature, for instance, is the ability to group related shapes or objects into a compound object which can then be moved, transformed, selected, etc. as easily as a single object.” This is just one of the enormous benefits of scene graphs, and one which would itself be a sufficient argument for using this concept in development.

One of OpenSceneGraph’s main goals is to make the benefits of scene graph technology freely available to all users. It was written entirely in Standard C++ and OpenGL, it makes full use of the STL and design patterns, and leverages the open-source development model to provide a development library that is legacy-free and focused on the needs of end users [8]. The core scene graph wraps around much of the OpenGL functionality, provides rendering optimizations and a growing set of add-on libraries which greatly speed up the development process. It frees the user from low level coding, and instead lets them focus on creating and controlling the content.

One of the key features of OpenSceneGraph for utilizing it in our project was its ability to manipulate a wide range of database formats. It does so via an extensible dynamic plugin mechanism - to date, the distribution includes more than fifty separate plugins for loading various 3D database and image formats. Some of the 3D database loaders include Alias Wavefront (.obj), 3D Studio MAX (.3ds) and the native .osg ASCII format [8]. The formats
listed were tested successfully with our application, but there should be no problem in using one of the other file types listed.

Using OpenSceneGraph had many advantages, but they came at a high price. Although it is one of the highest performing 3D graphics toolkits around, the literature on the subject is scarce, the tutorials are often just a little more than raw code, and many times there was no way to comprehend certain features of the library or detect a bug in the code, other than posting a topic on the official OpenSceneGraph forum and hoping for the best. Learning to use OpenSceneGraph was probably the hardest part of the project, but at the same time a rewarding experience, and certainly a great challenge. The following section explains exactly how it was used in our program.

3.3 Implementing the algorithm

As was said in Section 2.2, in order to find the best viewpoint of a 3D scene, we first need to examine the structure of that scene and find out what kind of information it contains.

In OpenSceneGraph, the scene graph tree is headed by a top-level root node. As stated in [9]: “Beneath the root node, group nodes organize geometry and the rendering state that controls their appearance. Root nodes and group nodes can have zero or more children. At the bottom of the scene graph, leaf nodes contain the actual geometry that makes up the objects in the scene.” This means that we can start at the root node, and make our way down to geometric information of primitives, such as dimensions and position in space. If needed, we could also build additional information into the scene, for example adding an arbitrary attribute to different objects, or even different primitives.

Next we need to establish a well-defined method for determining the quality of any given viewpoint, deriving only from the information available to us, which includes information added in the precomputation phase.

As mentioned in Chapter 2, we need to know how many different faces are seen in the scene, as well as the area that they cover. Here we face a problem of 3D vs. 2D. We can imagine standing at a certain viewpoint and taking a photo of the scene, such as the one in Figure 3.1. What we get is basically a two-dimensional piece of high-quality paper with different objects on it. Our brain is capable of comprehending the third dimension and differentiating the objects, but it is only because we know what trees, buildings or humans
look like. If we looked at a picture with unfamiliar forms and objects, such as the one in Figure 3.2, we could not be certain what the picture represents, or what the right way of looking at it was.

Figure 3.1: A real-world 3D scene

Figure 3.2: Pablo Picasso’s “Head of a woman”

In Chapter 2, we have discussed the two important parameters for computing the quality of a viewpoint - the number of faces seen and the total visible area of objects. Again, we take Figure 3.2 as an example. The computer “sees” the same picture that we see. How does it extract the information that it needs? How does it know which surfaces are visible in the
projection and to what extent? It needs a way to distinguish between different primitives in a way that is preservable when projected onto the plane of the screen. This can be accomplished by first stripping the objects of their textures (Fig. 3.3(a)) and assigning a different color to each primitive, saving one color (e.g. white) for the background (Fig. 3.3(b)). In this way, we can easily obtain the relative projected area of objects by considering the background pixels to be colorless, and counting all the colored pixels on the screen. We can also compute the number of visible faces by counting the number of different colors appearing on the screen. In this way, we obtain all the information necessary for determining the quality of a viewpoint by using one of the methods described in Chapter 2.

![Figure 3.3: Scene processing in precomputation phase](image)

Finally, having a set of viewpoints, we are able to find the one with the highest quality according to the method used. Given the primary focus of our application, we have decided not to let the user choose the set of viewpoints, but rather hard-code it into the program, as it would work in the same way for every other set of viewpoints.

We created a set that is thorough and equally distributed - starting with
an empty set and given the 3D scene, we took its bounding sphere, enlarged it and inscribed an icosahedron (Fig. 3.4(a)) in it, which is a regular polyhedron with 20 identical equilateral triangular faces, 30 edges and 12 vertices [11]. We added the vertices of the icosahedron to the set of viewpoints. Then we created a new point in the middle of every edge of the icosahedron (Fig. 3.4(b)), and projected each of those points onto the bounding sphere of the scene, so that all the viewpoints are at equal distance from the scene center. This is how the base set of viewpoints was created.

(a) A regular icosahedron  
(b) Viewpoint set before projection

Figure 3.4: Creating scene viewpoints

Additionally, a progressive refinement was implemented, where the best viewpoint of the base was found, and then the program searched for even better viewpoint positions in the area surrounding that viewpoint. This process is shown in Figure 3.5. Point A is the best viewpoint in the base set. All the points depicted also lie on the bounding sphere. Point B is the best viewpoint in the surrounding area of A, and it is better than A (Fig. 3.5(a)). Point C is the best viewpoint in the surrounding area of B, and it is better than B (Fig. 3.5(b)). Thus, C is the viewpoint returned by the algorithm. This is just to illustrate the process of progressive refinement. It could, of course, happen that A is the best viewpoint on the whole sphere, in which case the three points A, B and C would coincide.
It is also important to mention that every different measure for finding the best viewpoint implemented in the program can have a different set of viewpoints by overriding the method implemented as default. The process of doing this, as well as working with the internals of the program will be discussed in the next section.

3.4 Internal structure of the program

In this section we present the internal structure of the program, by outlining the most important classes, methods, data structures and programming concepts on which the implementation was based. A complete documentation of the program’s internals can be found on the accompanying CD. To differentiate between individual terms, we introduce writing conventions in the following way: programming concept names are underlined and each one is described in a separate paragraph; class names are typed in bold and each class is described in a separate paragraph; important methods of a class are introduced in a bulleted list at the end of the class paragraph and their names are typed in italics; important data structures of a class are introduced in a bulleted list at the end of the class paragraph and their names are underlined.

**BaseSolver** The abstract base class of all the implemented methods for finding the best viewpoint (i.e. solvers). It contains the implementation of
the algorithm described in the previous section, as well as virtual functions for creating the base set of viewpoints and the refinements of that set. This makes it straightforward to work with an arbitrary base set of viewpoints in derived classes (solvers), and leaves space for future modifications of the program in such a way, that the user could choose between different patterns of viewpoints without modifying the source code.

- **computeCameraPositions** A virtual function that creates the base set of viewpoints to be used in computation. Its default implementation is described in the previous section. It can be overridden in the derived class, giving an arbitrary choice of viewpoints to be used by the method implemented.

- **pinpointCamera** A virtual function that implements progressive refinement of the viewpoints, described in the previous section. Its default implementation is depicted in Figure 3.5. It can be overridden in the derived class, effectively changing the refining set of points used.

- **computeViewpointQuality** A pure virtual function that serves as a placeholder for implementation of individual methods for computing the viewpoint quality. The derived classes effectively implement the methods described in Sections 2.3-2.5.

- **computeBestViewpoint** A virtual function that finds the best viewpoint in the given set of viewpoints. It uses the previous function for individual evaluation.

- **handle** A virtual function that brings together all the pieces of the algorithm described in Chapters 2.2 and 3.3. It works directly with the underlying scene structure, finding nodes of interest, positioning the camera at the coordinates specified, managing the calls for updating the user interface and issuing calls to the function which saves the screenshot to file.

**Callback** A callback is a reference to executable code, or a piece of executable code, that is passed as an argument to other code. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer [12].
Visitor pattern The visitor design pattern is a way of separating an algorithm from an object structure it operates on. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures [13].

GeometryPaintVisitor A class based on the Visitor pattern, which prepares the scene for the main algorithm execution by traversing the whole scene graph and painting the individual primitives. Although the computer can register even the slightest difference of the two pixel colors, it would be very hard for the user to distinguish the objects in the scene, so this class takes the total number of primitives in the scene, and calculates a spread of colors, so that the whole spectrum is used.

HeuristicMeasureSolver A class derived from BaseSolver, which implements the computeViewpointQuality method, and can override one or more of its parent’s virtual methods. It already contains the two weight parameters introduced in Chapter 2.4, both with the default value of 1.

HUDCallback A class based on the Callback concept, which, upon selecting a method to be run, regulates the display of the method’s name and time elapsed in the user interface.

MainViewer The main class, which loads the scene from file, registers the solvers, assigns them as key-press handlers, creates the user menu and shows the scene on the screen.

- registerSolver Registers the solver with the program and assigns it as a key-press handler.
- createHUD Creates the menu for choosing the algorithms and shows their running time.
- run Prepares the scene and the user menu, and creates the loop in which the program is running and responding to user input.

SceneInfo A class containing general scene information, such as its background color and the total number of primitives it contains. It is very useful in the computation of viewpoint quality, as it enables us to differentiate
background pixels from the object pixels, and calculate the percentage of visible faces.

**ViewpointEntropySolver** See HeuristicMeasureSolver.

**SnapImage** A class which, upon request from BaseSolver, processes the current camera view, retrieves the information about the pixels shown on the screen, creates ViewpointInfo and SceneInfo objects, and sends them to BaseSolver’s main algorithm function for further computation. It also provides an interface for saving the screenshot to file.

**ViewpointInfo** A class which contains information about the total area visible in the scene from a given viewpoint. It serves as a carrier between the lower-level SnapImage and the more abstract BaseSolver class.

- **colorsCount** An STL map data structure, containing pairs (color - number of pixels with that color). It is used to store information from the current screenshot, and supports fast insertion and searching, which improves the overall performance of the main algorithm.

**WeightedHeuristicMeasureSolver** A class derived from HeuristicMeasureSolver, which sets the two weight parameters and delegates the main part of the computation to its parent class.
Chapter 4

Results

In this chapter, the terms Heuristic measure, Viewpoint entropy and Weighted heuristic measure will be used extensively. For practical reasons, we will refer to them by their respective initials HM, VE and WHM. For WHM, we will also write the values of the two weights $W_1$ and $W_2$ from the formula

$$C(V) = W_1 \sum_{i=1}^{n} \left[ \frac{P_i(V)}{P_i(V) + 1} \right] + W_2 \sum_{i=1}^{n} P_i(V)$$

(introduced in Chapter 2.3), so that, for example, WHM1-3 indicates the case where the projected area of the scene is three times more important than the number of visible surfaces.

The resulting outputs of these methods depend heavily on the discretization of each model’s surfaces. The same surface can, for example, be represented both as a one big triangle, as well as a number of small triangles. This difference could manifest itself in the final output if, for example, the method used is highly focused on the number of faces seen. That also means that the scene with a low level of detail could produce completely different results from the high-detailed version of the same scene.

By the very nature of the problem, we cannot say that there is a right and wrong way of choosing the best viewpoint; one method can give better results for certain models than other, and we can only discuss the rightness of choice from an aesthetic, subjective point of view. With that said, we show the outputs of the program given one of the models from the accompanying CD as input, and discuss the results considering the information that we have about the methods used.
Figure 4.1 shows a truck model with the cabin comprised of a number of small polygons, while the cargo area of the truck is made of comparably less polygons of greater size. Considering the number of polygons and their average size, the bottom part of the truck lies somewhere in between the cabin and the cargo area, being moderately discretized and comprising of middle-sized polygons.

![Highly discretized cabin](image1.png) ![Moderately discretized bottom](image2.png)

(a) Highly discretized cabin  (b) Moderately discretized bottom

Figure 4.1: Discretization of a 3D truck model

In Figure 4.2, we see the outputs of methods HM and VE. HM balances the highly discretized regions of the cabin with the less discretized cargo area, while VE focuses on the cabin, but also shows the fairly discretized bottom of the truck, seeing a little less faces than if it were focused on the cabin only, but also covering greater projected area of the object.

Figure 4.3 shows the outputs of method WHM for cases when we assign more value to one of the two selection criteria. The output in Figure 4.3(a) is, as expected, focused on the highly discretized cabin, while Figure 4.3(b) supposedly shows the viewpoint with the largest projected area of the truck. Notably, all of the viewpoints nicely present the three-dimensional nature of the object, but that is a feature that depends heavily on the discretization of the object, and as such does not stand as a rule.

Looking closer at these outputs, we can see a strong correlation between Figure 4.2(a) and Figure 4.3(a). We could try to find out if this was just a coincidence or there was a reason for such similarity. In Chapter 2.3, we stated that both HM criteria can have values between 0 and 1, and that they have the same impact on the final result. This is only true in theory, but it does not stand in our program. The cause lies in the implementation
Figure 4.2: Heuristic measure vs. Viewpoint entropy

Figure 4.3: Weighted heuristic measure with different weight proportions
of camera positioning in the scene. All the viewpoints that are being tested are positioned at an equal distance from the scene center, calculated in a way which guarantees that all objects of the scene lie in the camera’s field of view. This calculation makes it effectively impossible for the objects in the scene to cover more than approximately 60% of the screen at any point during algorithm execution. In addition, the projected area of the scene can never approach zero, thus significantly reducing the range of the second parameter. On the other hand, the first parameter truly does take on values from almost the whole (0, 1) interval. As an example, it suffices to think of a scene with a single pyramid in it. When looking from the top, all the faces are visible, except the base. When looking from beneath, only the base is visible. Based on the fact that their effective value ranges differ considerably, we can conclude that the two parameters do not contribute equally to the final result. We can confirm this by looking at Figure 4.3, which represents best viewpoints for WHM3-1 and WHM1-3. We have tested the WHM method with a range of different weight ratios, and the results for WHM100-1 and WHM1-100 look almost exactly like Figure 4.3. This means that the two viewpoints in 4.3(a) and 4.3(b) truly represent positions with the greatest value. Figure 4.4 is identical to Figure 4.3 and it shows the outputs for WHM1-1.9 and WHM1-2.

![Figure 4.4: Bias towards the number of faces in WHM](image)

If the two parameters did have the same impact on the final output, we would see some kind of shift as soon as the second weight outgrows the first. Nevertheless, for $W_2$ in the interval $< 1, 2)$ there is no visible change. It is only as $W_2$ becomes twice as large as $W_1$ that the change in position occurs.
This, of course, is only applicable to the model at hand and not as a global rule, as once again the discretization of the models plays a great role in the final output.

Another thing worth noting with all the models tested with WHM method is that after the proportion is established to favor one of the two parameters (e.g. $W_2 \geq 2$ in Figure 4.4), there are minimal differences in the output upon enlarging the difference between the weights (e.g. WHM1-3 vs. WHM1-100).
Chapter 5

Conclusion

The main goal of this thesis was to implement several methods for finding the best viewpoint of a 3D scene and assess them in a simple experimental application developed for that purpose. After an introduction to the concept and purpose of tackling this problem in the first chapter, in the second chapter we presented some of the earlier approaches to the problem and introduced the algorithmic concept of our approach, together with the methods for computing the quality of an individual viewpoint, which were implemented as a part of this project. In Chapter 3, we explained some of the design choices made during the development process, described the graphical library used in the program and elaborated on the internal structure of the application. Next, we presented the outputs of the program, some interesting facts that were discovered about the implemented methods, as well as some aspects of the algorithm structure that can be approached differently.

We see the main contribution of this project in its role as an open-source demonstration tool, which turns raw theory into something vivid and tangible. As stated in Chapter 4, by the very nature of the problem, we cannot say that there is a right and wrong way of choosing the best viewpoint, but our program can certainly help in discovering the patterns of good and bad behavior of different methods, and it does so in a much more natural and straightforward way than, for example, examining a certain formula and predicting special cases for which it fails. We have been unable to find a similar program on the Internet, as they are mainly developed by the pioneers in this particular area of research, and used internally for purposes of testing and tweaking.

As a part of the project, we have created a modification of an existing
method for finding the best viewpoint of a scene, and though the modification was not drastic, it gave some interesting results which exposed certain characteristics of the original method.

In the process, we have learned a lot about computer graphics and programming in general, and although there is still a vast amount of space for improvement, we sincerely hope that this project can help in understanding the concepts here described, and serve as an encouragement for people who are new in the field, to delve in and learn by doing, as there truly is no better way to success.

5.1 Future work

As stated in Chapter 1.2, the main focus of this project was on implementing the methods for computing the best viewpoint of a 3D scene, and the program for examining the performances of the implemented methods was to be kept simple (for the end user, at least) and serve its purpose. There are several aspects of this program that could be improved in the future, the first and most obvious being the user interface. At present, the user is unable to do anything but choose the method to be run. A modified user interface could, upon choosing the method to be run, include selecting the camera placement pattern for the base set of viewpoints, as well as its density, choosing whether the scene should be evaluated only “from the ground up”, or from the bottom hemisphere as well, deciding whether the best view of the scene should be saved or not, and at what location, etc.

Another interesting modification would be turning the methods into standalone DLL files, which could be used in the program just by placing them in a certain location on disk. This modification, though simplifying the process of adding new methods, would demand significant, time-consuming changes to the internal structure of the program.
Bibliography


Appendix A

CD contents

The accompanying CD has the following structure:

- **doc/**
  - doxygen/ - API documentation
  - Readme.txt - A brief user guide
  - GPL.txt - License

- **exe/**
  - models/ - models of 3D scenes
  - BestViewpointFinder.exe - main executable
  - Test.bat - batch file for convenient testing of different models
  - Necessary DLLs

- **src/**
  - BestViewpointFinder/ - Visual Studio 2008 solution

- **txt/**
  - latex/ - LATEX sources of this text
  - Thesis.pdf
Appendix B

User guide

B.1 System requirements

The program should run on any computer with Windows XP or newer, which has Microsoft Visual C++ 2005 Redistributable Package installed. It was tested successfully on Windows XP SP2, Windows Vista and Windows 7. Saving image files also requires that the user of the operating system have write permission for the folder containing the executable. Running time of the algorithm on Intel Core 2 Duo 2GHz processor with 4GB RAM is approximately 12 seconds on average (depending on the scene used). The program works on single-monitor displays only.

B.2 Usage

The files needed for running the program can be found in the exe/ folder on the accompanying CD. One way of running it is by using the BestView-pointFinder.exe file, which takes as its arguments the name of the file containing the 3D scene, and the two weight parameters (float values) for the Weighted heuristic measure algorithm. The weight parameters are optional and both have a default value of 1.

Since typing the names of different models is tiresome, a short script file, Test.bat, was prepared. It contains eight different scenes from the models/ folder, so instead of typing in long filenames as the first parameter, the user just inputs a number in the interval [1..8]. The last two parameters are the same as for the executable. Both ways of starting the program require typing
the arguments in the command line. The two commands are equivalent, and
their respective forms are the following:

- BestViewpointFinder “models\dumptruck.osg” 3 1
- Test 6 3 1

Upon starting the program, the first parameter is validated, and the 3D
scene is loaded and shown on the screen. The user can then explore the
scene by rotating, panning and zooming with his mouse. A list of algorithms
currently implemented is shown in the lower left corner of the screen, and
the user can choose between them by pressing the respective keys on the
keyboard.

When a key associated with one of the algorithms is pressed, the com-
putation starts, and in the lower left corner the user can keep track of the
elapsed time. Finally, the camera is positioned to the best viewpoint of the
scene, according to the respective algorithm. In addition, if the user of the op-
erating system has write permission for the folder containing the executable,
a PNG image file is generated (having the same name as the algorithm that
created it), containing the best view of the scene. If the containing folder
is write-protected, the shell displays a warning message, and the program
continues. After the camera is positioned, the user regains control of the
scene, and can continue exploring it, or start a different algorithm. When
the same algorithm is run more than once, the placement of the camera is
momentary.