FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Filip Rechtorík

Creati - Game Development Platform

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D.

Study programme: Computer Science
Specialization: General Computer Science
Prague 2021

Title: Creati - Game Development Platform

Author: Filip Rechtorík

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The goal of the thesis is to create a game engine based on entity component system. The game engine should include editor, which would allow developers to create their own components, link custom assets to these components, create world from these components, and have them visualized in the editor. Editor will also provide tools for real-time debugging. By default the engine should contain graphics system, input system, simple physics system, and user interface system. As a final step the engine should support exporting the project as a standalone game.

Keywords: Game Engine 3D Rendering DirectX .NET

Contents

1. Introduction

1.1. What is a game engine?

A Game Engine is defined as being a set of software tools or API's built to optimize the development of a video game. This will typically include a game loop or at the very least a 2D or 3D rendering engine.
— https://gamescrye.com/blog/what-is-a-game-engine/ [1]

Creating a game from scratch is a lot of work. The programmer has to create a rendering system, handle input, figure out user interface, create a world editor or instantianiate the whole world using text. It also requires a lot of additional knowledge about rendering, working with the GPU, and general game engine architecture. For that reason, it is standard to use an existing game engine when creating a new game.

We would like to create a game engine that can compete with modern game engines such as *Unity* and *Unreal Engine*. The game engine should be fully modifiable by the developer and should also include novel features such as *CLSL*, which stands for *C# Shading Language. CLSL* would then allow developers to write shaders for DirectX directly in *C#* rather than having to learn and use *HLSL*.

Modern game engines include an overwhelming amount of features, far exceeding the scope of this thesis. For that reason we chose to create a usable core of a game engine with an editor that can later be expanded on by a follow up thesis (for example the *CLSL*).

1.2. Goals

1. Create a fully extendable game engine core with the following features:
    a. Extendable game object system
    b. Ability to create new systems
    c. Extendable rendering system
    d. Extendable physics system
    e. User interface system
    f. Input system
    g. Asset cache supporting custom resource types
    h. Saving and loading of scenes
2. Create an editor with sufficient features for basic game development allowing us to:
    a. Create and edit scenes, game objects, and components
    b. Debug the game and inspect the properties of our game objects and components while debugging
    c. Easily define a custom user interface for editing components
    d. Link resources and game objects
    e. Create custom resources from classes
    f. Write custom scripts that can extend any part of the game engine
    g. Export the game as a standalone app

2. Implementation Analysis

2.1. Programming Languages

First we need to choose programming languages for the project. We need a programming language for the game engine itself, a programming language for the rendering system, and a programming language for scripting. The languages we will consider are: C#, C++, Rust [2], and Lua [3].

The first language, *C++*, is a very high performance programming language with manual memory management and access to a huge amount of libraries. The C++ is however slower to develop in because it is a lower level programming language. The language is also unsafe because it is easy to create dangerous bugs such as memory leaks.

A safer, but still a very high performance language is *Rust*. *Rust* is a newer programming language similar to C++ but significantly safer. Because the language is newer, it doesn't have as many libraries as C++.

C# is a programming language with a pretty good performance, although not as good as C++. One of the reasons why C# is slower than C++ is because it uses a garbage collector. Garbage collector can be problematic in games because it can cause stuttering when the garbage collector is collecting. The compiler also cannot perform as many optimizations as for C++. However, C# is a very mature programming language with many libraries. *C#* is also the only language from the ones mentioned that supports reflection.

Lastly we have a scripting language *Lua*. From the languages mentioned, Lua is the easiest to learn. However, because it is a scripting language, the performance is significantly worse than that of the other languages.

Because we will need to inspect arbitrary game objects, we pretty much need reflection, so C# is an easy choice for both the core engine and scripting. The rendering engine could be in either C#, C++, or Rust. All of the languages have libraries for working with the GPU (DirectX or OpenGL). If we chose C++ or Rust, we would have a slightly higher performance. However, by using C#, we can integrate the rendering code more directly in the game engine's core. The author is also most familiar with this programming language which is why it has been chosen for the rendering code.

2.2. Game objects

Pretty much all games are composed from a number of game objects. Take a game like Portal 2 [4] for example.

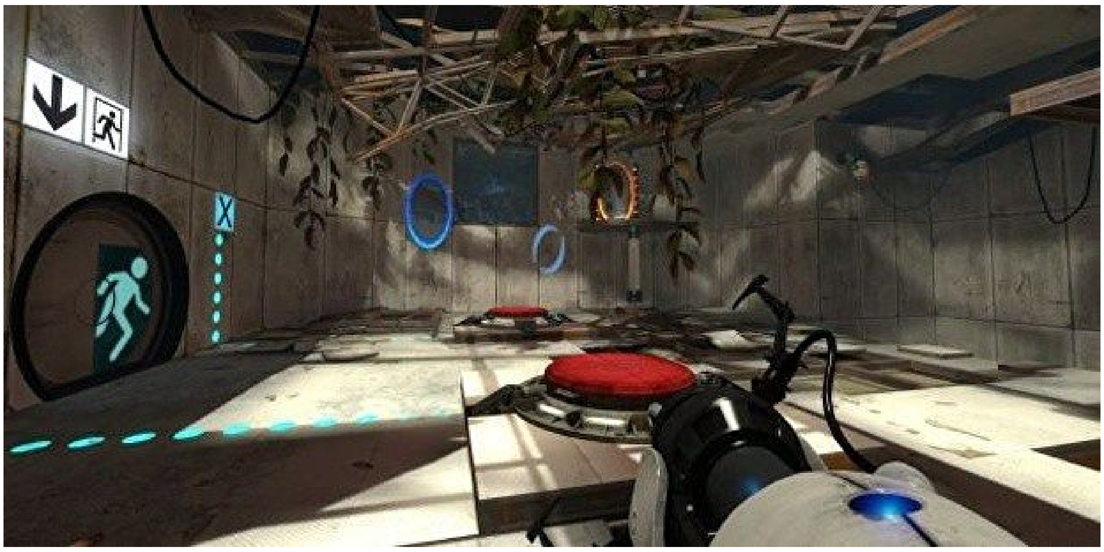


Figure 2.1: A screenshot from game Portal 2

In the figure 2.1, we can see several game objects. Player is a game object. Each of the buttons is a separate game object. Even the portals are their own game objects.

In games, different game objects can have different properties. A patch of grass on the ground is only a graphical game object and the player can pass through it. A house is a game object that cannot move but is rendered and other objects can collide with it. A car is a game object that is rendered, other objects can collide with it, and is also affected by physics. The player character is also controlled by pressing keys on the keyboard. On top of that, some game objects will need to have scripted behaviors.

To deal with those arbitrary game objects, there are two main strategies – inheritance and composition.

Inheritance works by having some base class, e.g. *GameObject*, which we then extend by the functionality we want. The *GameObject* class typically only knows its position in the world. When we want our game object to get rendered in the world, we would create a new class *GraphicalGameObject*, inheriting the original *GameObject*. Then if we wanted the *GraphicalGameObject* to behave physically, we would create a class *PhysicalGameObject,* inheriting the *GraphicalGameObject* and implementing the physical behavior.

This approach is however quite problematic. As more and more functions the game objects can have, the hierarchy becomes less and less clear, and eventually forces the developer to either duplicate code or to create a diamond dependency [5]. On top of that, diamond dependency is not even supported in C#.

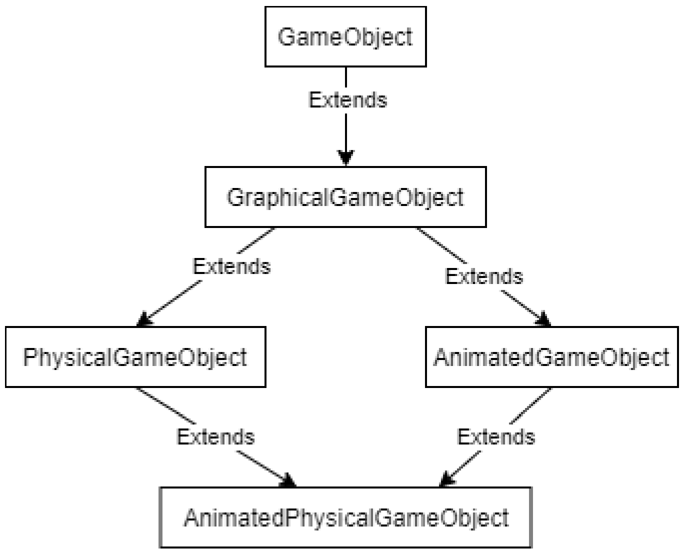


figure 2.2: Game objects creating a diamond hierarchy

For example, consider the class hierarchy in figure 2.2. We can clearly see that it creates a diamond dependency. If we wanted to avoid it, we could for example extend only one of the classes (e.g. *PhysicalGameObject*) and then implement the whole functionality of the other subclass (*AnimatedGameObject*) all over again, duplicating code.

An alternative approach to this is the composition. In the context of game engines, composition is commonly called the *Entity Component Framework* [6]. The *Entity Component Framework* has a base class called *Entity* or *GameObject*. This class contains a collection of components. Each *Component* gives a specific functionality to the game object. For example, to create the problematic *AnimatedPhysicalGameObject*, we would simply create a new game object with components *Mesh*, *MeshAnimator*, and *PhysicalBehavior*. For physics, there would also be some collider component, defining the physical shape of the game object for collisions.

The *Entity Component Framework* is clearly a much cleaner and extensible approach, so we decided to use that approach in our game engine.

In our game engine, game objects are stored inside a class called *Scene*. We can freely switch between multiple scenes – for example from *Main Menu* scene to the *Level 1* scene. Each *Scene* then contains a collection of game objects. Each *GameObject* is then composed of many components, giving it functionality. Furthermore, game objects can create a hierarchy. If one game object (e.g. player) is under a different game object (e.g. ship), then moving the top game object will also move the game objects under it. This means that the player would move together with the ship.

2.3. System creation

A system is a collection of classes that give functionality to a set of components. We will need to create several systems throughout the game engine and we need a unified strategy for creating them. There is going to be a graphical system, physical system, input system, user interface system, and then user defined systems.

For a system to give functionality to the components, it must know about them and hold them in some collection. All systems also need to distinguish between different scenes because game objects from different scenes should not affect each other in any way. All of the systems will also need to be updated in a specific order. For example, we need to update physics and move our objects before we can render them.

There are three options for creating systems.

1. Have the system be a component
2. Create a special collection of systems inside each scene
3. Make the system a static class and remember the collection of components per scene

To make the following text more clean, we will assume our system is called *Physics* and our components are called *RigidBody*.

In (1), we would create a component called *Physics* that would manage *RigidBodies*. Assuming that every component has some *Update()* method called every frame, we would override that method and do the calculations there. We would then call something like *Scene.GameObjects[“Physics”].Component<Physics>()* to access our system. The problem is that there is no clear way to make sure the components get updated in the correct order.

In (2), each scene would have a collection of systems. The systems could be lazily initialized when we first create them. To access it, we would call something like *Scene.System<Physics>()*. Each system would have a virtual method called *Update()*. To make sure the systems get updated in the right order, we could have our game loop retrieve the systems in their respective order and call the *Update()* on them.

In (3), we would have a static class *Physics* containing the collection of *RigidBodies*. Accessing the static class would be super easy. To update the systems in the right order, we just have to call their respective *Update()* methods in the game loop. The only problem here is that *RigidBodies* from multiple scenes get mixed together. To solve that, we will have to keep multiple collections of *RigidBodies*, one per each scene. Doing that in every system, and especially user defined systems, would be quite annoying. We can however use a trick to make this process easier. We can create a class[1] that holds multiple copies of the same type and always gives us the one for the current scene.

We will definitely not use strategy (1) because of its ugliness and the difficulty with ensuring the correct update order. Strategies (2) and (3) are both pretty much equal in their effectiveness and the choice comes down to preference. The game engine uses the strategy (3).

[1]In the game engine, this corresponds to the generic class *SceneStatic<T>*.

Component collections

Every system needs to have at least one collection that holds all the relevant components. We will now discuss the problematic of adding and removing these components from the collections.

When we create a new component, we have two options of how to add it to the relevant collection.

1. Let every system react to some event called every time a component is added or removed.
2. Let the components add themselves to the systems.

If we use strategy (1), then every time a new component is created, every system will react to the event and check if the component belongs there. This will clearly create some unnecessary overhead every time components are created.

The strategy (2) on the other hand doesn't have any overhead. It simply accesses the collection through the static class of the system.

However, removing components using this strategy is problematic. If we (or the user) forget to remove the dead component, then the dead component will remain in the collection and might interfere with the other objects there. It isn't even possible to detect that and throw an exception to notify the user. This problem is solved in the game engine by using a smart collection, which checks if the component is dead every iteration, and if it is, it removes the component from the collection.

2.4. Game Loop

Every game has a game loop at its core. An example of a simple look is in the figure 2.3. This loop processes the input, updates the game, and draws the world. Real game loop would be more complicated and have more items. We will use our game loop to update the systems in the correct order.

```
while (running)
{
    Input();
    Update();
    Draw();
}
```

Figure 2.3: Simple game loop

Having a game loop similar to the one in the figure 2.3 would be very inflexible which goes against our goal of extensibility. Since systems can be created by the user, we require that those systems have the access to the same system creation tools as the integrated systems.

In other words, we require a dynamic game loop that can be modified by the user. To do that, the game loop in the game engine is stored as a list of actions (class *Action* in C#) representing that loop. We can then modify the game loop by calling *Loop.ModifyGameLoop(...)* method, which gives us access to that list.

The default game loop looks like this:

1. Manage game objects[1]
2. Update game time
3. Update input
4. Call *EarlyUpdate*[2]
5. Call *Update*[2]
6. Update physics
7. Call *LateUpdate*[2]
8. Render the scene
9. Present the scene in the window
10. Update WPF

The game loop in the game engine uses cooperative multitasking [7]. That means that each method is expected to perform one step and then return the control to the main program.

2.5. Rendering

For rendering, we can either use an API that communicates directly with the GPU (like DirectX or OpenGL), or we can use an existing rendering engine (like OGRE [8] or Urho3D [9]).

Using an existing rendering engine would make the writing of the graphical system much easier. It would also give us a lot of features for free. However, all external libraries store the information about the scene in their own structure. That means that we have to keep updating the that to match the scene in the game and that would cost us performance. Not only the cost of having to update the position, rotation, and scale of each object every frame, but also by making it more difficult for the cache to keep up as we keep accessing memory in many different locations. The bigger issue is that keeping the rendering structure of the library udpated while editing the corresponding component in the editor can be difficult and lead to unexpected issues. Using an external library could also lead to problems later on, when some particular feature is not supported in the library[3]. For the reasons mentioned, the graphical system will use either DirectX or OpenGL. Since the game engine is not cross platform, we are not forced to use OpenGL. The author of this thesis is familiar with DirectX, so that is the API used for the graphical system. The library used for working with the DirectX from C# is *SharpDX* [10]

[1]This first activates newly created game objects and calls the method *Start()* on their components. Then it removes all destroyed game objects and calls the method *Destroy()* on their components.
[2]*EarlyUpdate*, *Update*, and *LateUpdate* are events that the components can subscribe to to define their behavior.
[3]Neither OGRE nor Urho3D allow us to use DirectX API directly.

Displaying the render output

The common way to display the resulting image from our rendering in the window is by connecting it with a *Form* from Windows Forms [11]. It is also the only well documented way.

However, there also exists a way to connect SharpDX with WPF using the *WPF DirectX Extensions* repository [12]. It is not well documented but it allows us to display our renders in the WPF window, which in turn allows us to use the powerful WPF system to create user interface both in the editor and in the game.

Rendering pipeline

There are two main rendering strategies for rendering using DirectX, forward rendering and deferred rendering [13]. While forward rendering is more straightforward and allows for better performance, the deferred rendering enables us to render an unlimited amount of lights. The reason for this is that in forward rendering, we can only put a limited amount of lights inside pixel shader. On the other hand in deferred rendering, we can calculate the physical properties of each pixel first and then calculate all the lights one by one. But that doesn't mean that deferred rendering is simply superior, forward rendering is better suited for graphical objects that do not depend on light and is used to create special. Since there is a use for both approaches, the graphical system implements both.

In the rendering, the game engine first performs *surface rendering*, which renders the surface properties on the so-called G-Buffer [14]. The G-Buffer is a set of textures, each for one property of the surface. The G-Buffer remembers the position, color, normal vector, emission color, alpha, and reflection properties of each pixel. Then the *light rendering* uses the G-Buffer textures as its input and calculates the effect of each light on each pixel as a color and adds them all together. We perform the *forward rendering* last because we want to also use it for special effects that might override some parts of the result from *light rendering*.

The surface rendering is used for all the rendering of meshes while the forward rendering is used for things like skyboxes and particle generators.

Performance

We also need to consider the performance of our rendering. One thing that affects rendering a lot is repeatedly setting the state of the GPU. Imagine if we are rendering two objects with the same material. If we were to simply render them one after another, we would set shaders, textures, constant buffers, samplers, other DirectX states, and then call the DirectX *Draw()* [15] method. We would then do the same thing for the other object. All of those are API calls to the GPU which are very expensive. The important thing is that those secondary calls are completely unnecessary and if there are even more similar objects, the number of unnecessary expensive API calls will grow even more.

One way to solve this issue is to hold a copy of the GPU's state in the RAM and only perform the API calls when an actual change is made. This would work well if it wasn't for the fact that the objects are not obligated to be sorted well. The objects with shared states might simply not be next to each other.

Another way to solve this is by creating some sort of material system. A material system would group objects that share some rendering calls together under a material. Typically a material is split between its *material type*, which holds the state changes common between all instances of that material, and then a *material instance*, which holds the state changes only for this particular instance of the material. Each graphical object would then be composed of its material and some *Render()* method which would usually just set the mesh and call the DirectX *Draw()* method. The problem with this material system is that it is quite inflexible. Who says there only have to be 2 steps in the material?

Rendering graph

For this reason, the game engine uses a special material system where materials can be composed of any number of *rendering steps*. Each such rendering step can then be entered and exited. On entering, it sets the necessary DirectX states and on exiting it reverts some of the them[1]. The rendering steps are kept inside a tree structure called *RenderingGraph*. This material serves as a user-defined heuristic because the developer can split the rendering process into parts they thinks are likely to be shared between different objects or groups of objects.

To understand the rendering graph, we look at the figure 2.4 where we see two objects with different rendering steps (yellow and orange). After we insert both of them inside the rendering graph, we can see that they form a tree by sharing the first step (figure 2.5). In the figure 2.6, we can then see how the rendering process works. The green arrows mean we are entering the state on the right and the red arrows mean we are exiting the state on the right. The order is indicated by the numbers on the arrows.

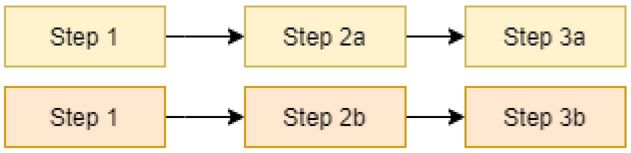


Figure 2.4: Two graphical objects sharing the same first step

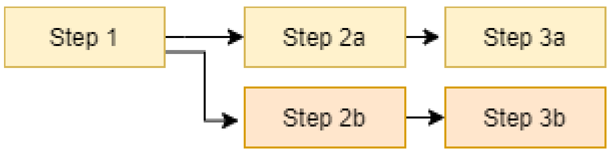


Figure 2.5: Two graphical objects with shared first step inside the RenderingGraph

[1]Not all changes need to be reversed. For example, setting a pixel shader doesn't need to be reversed because the next material is guaranteed to override it.

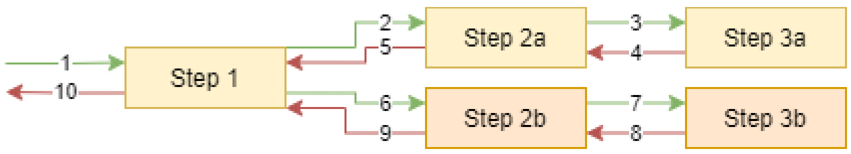


Figure 2.6: The rendering order of the RenderingGraph

The rendering steps might look like this:

1. Set material type
2. Set material instance
3. Set the geometric data
4. Draw

The material type step (1) would set the shaders and one of the textures for example, then the material instance step (2) would set the remaining textures and constant buffers with the parameters. The geometric data step (3) would set the vertex buffers and index buffers. And lastly, the draw step (4) would call the DirectX *Draw()* method.

Each of the individual renderers uses a rendering graph to hold the objects we want to render. The graphical objects added into the rendering graph are defined as a set of rendering steps.

Rendering extra information inside editor

Sometimes we would like to render a little bit more in the editor than in the game. We would like to see the shapes of our otherwise invisible colliders, some indicators for where the invisible light source is, the shape of camera frustum, and so on.

In other words, we would like an extra pass inside our rendering pipeline that gets only performed when we are in the editor. This is achieved by the combination of two things. First, a special *EditorRendering* class which contains a rendering graph for the editor-only graphical elements. Second is the special *EditorCamera* which has a modified rendering pipeline to include the *EditorRendering*. Because the special gizmos we want to draw might want to overlay the result of the other rendering, we render the *EditorRendering* last.

Whenever a component wants to render special information in the editor, it creates a rendering program from the rendering steps and adds it to the *EditorRendering*'s rendering graph.

We would also like to display the extra information conditionally. For example the camera preview and camera frustum (figure 2.7) should only be visible when the game object containing that camera is selected. If it wasn't, it would create a lot of visual clutter inside the editor. Other information, such as colliders, we would only like to see when we are hovering with our mouse over the particular component.

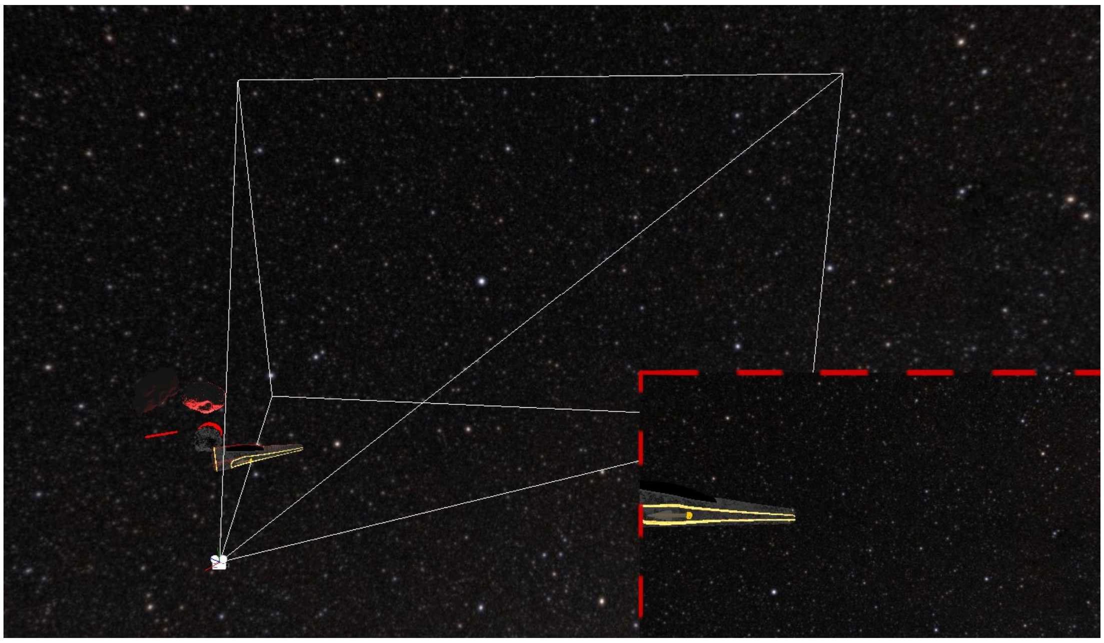


Figure 2.7: Camera frustum being rendered inside the editor

Shaders and rendering programs

A rendering program is a set of shaders making up the rendering process.

We need a consistent way across the game engine to access shaders and rendering programs. Only with a good and consistent design can we avoid thinking about the particular shader implementations while using the shaders.

First, shaders. All shaders reside in a static class with the name of the shader prefixed by “Ps”, “Vs”, “Gs”, “Hs”, or “Ds”, depending on the type of shader. Each shader class then contains a method *Set(...)*, taking a single parameter of a DirectX *DeviceContext* [16]. This method sets up the shader on the GPU so it can be used. Second method is *SetParameters(...)*, which takes arbitrary parameters depending on the particular shader and sets them on the GPU. The *Set(...)* method is meant to only be called once while the *SetParameters(...)* method can be called multiple times, for example per material instance. Some shaders can also contain the method *Reset(...)*, which resets some of the DirectX state to avoid interference.

On top of shaders are built rendering programs. All programs should reside in a singleton class with the suffix “Program”. This class must inherit the *RenderingStep* class which allows it to be used as part of the rendering process. Whenever the rendering step is entered, it should call the *Set(...)* method of all the shaders it uses, plus potentially some additional setup. Whenever the rendering step is exitted, it should call *Reset(...)* on all the shaders that have the method and reset whatever other DirectX states it might’ve set on entering.

The rendering program then also includes a *SetParameters(...)* method which then transitively sets the parameters for all of the shaders. Optionally, the program can also contain an *Instance* subclass, also inheriting the rendering step, which encompasses all the parameters and sets them on entering.

2.6. Graphical components

In this chapter we will discuss the design of existing graphical components.

When creating graphical components, we will almost always want to use two special rendering steps. First is a rendering step that sets up the geometrical data. For this reason, the game engine includes the *GeometryDataStep*. Second is the last rendering step which calls the DirectX *Draw()* method.

The last rendering step is actually even more interesting, because in order to remove the destroyed components from the rendering graph, the last step has to know if the component has died. At the same time, the last rendering step tends to have a different implementation per component. Since the implementation differs per component, we would like to use either a lambda function or a method inside the component. If the method is inside the component, we can find the component from it and then figure out if the component is alive or not. Because of that, the game engine contains a special rendering step called *DrawMethod* which takes a method inside the same component as its only parameter. When the component later dies, the *DrawMethod* uses its reference to the component to notify the rendering graph that it should get removed.

Mesh

Meshes are the most common thing rendered using the game engine. For that reason, we would like to make mesh rendering as simple as possible.

First thing we can simplify is the way we set up our vertex buffers and input layout [17]. Different vertex shaders need different buffers to be set up in a different layout. However, by using reflection on the vertex shader we can deduce what the desired input layout is. We can then use that information to extract the vertex buffers from the mesh in the correct order. In practice, this is split into two parts. First, when we load our vertex shader into *MeshVertexShader*, the analysis is done using reflection and a *Descriptor* is generated. *Descriptor* is a lambda method that takes as input the mesh and returns the list of vertex buffers in the correct order. Whenever we are setting our vertex shader, we have to set the current descriptor inside a static variable. The second part is inside the rendering step called *MeshInput*. This rendering step gets initialized with a mesh and then sets up the mesh vertex buffers correctly on the GPU using the current *Descriptor*.

The last thing we can do is to create a flexible but simple to use material system. Each material corresponds to all of the settings of the rendering pipeline [18] with the exception of the geometrical data. We want to use the materials assets so the base material class is marked with the *[AssetClass]* attribute. A material is just one instance of a class, however, it logically corresponds to two rendering steps. This is apparent because each material instance will have some DirectX settings that are shared between all instances (e.g. shaders), while other data like textures will depend on the particular instance.

To turn one class into two rendering steps, we have done two things. First thing is that we made the material class be a rendering step in itself. Second thing is that we added an abstract getter returning a rendering step that sets the shared part of the material settings (called *Program*). We will typically use a singleton to define the program.

The game engine includes a component *MeshRenderer* which uses this material system. In particular, the rendering steps to render the mesh are:

1. Material program (shared part)
2. Material instance
3. *MeshInput*
4. *DrawMethod*

There are 2 materials provided in the game engine but the user can easily create more by extending the *Material* class. The first material provided is the *SimpleMaterial*, which only uses color and emission. The second material is called *ComplexMaterial* and uses textures for color, normal, binormal, emission, and reflective properties.

Camera

Every rendering system naturally needs cameras. By having the camera be a component, we can move it together with game objects (for example follow a car). What properties should a camera have? The obvious properties are field of view, view and projection matrices used for rendering, and perhaps a background color. Then we might want to have a viewport rectangle so that we can render the camera only on a part of the screen. This viewport rectangle needs to have a relative size (from 0 to 1) so that it scales together with the window resizing.

A question is what to do if there are multiple cameras? The game engine solves this by first allowing every camera to either be active or inactive (which can be used for switching cameras) and then by giving each camera a *Depth* property. The cameras are then rendered in such an order that the shallowest camera is on top.

Sometimes we also want to use a special rendering pipeline and the camera is an intuitive place for it. By replacing the rendering pipeline, we can change how and which objects get rendered. For example in the editor, we add an extra step that renders editor-specific graphics.

Image renderer

Inside the editor we needed a way to render 2D pictures in 3D space. This functionality was necessary to show floating icons for lights and cameras to indicate their position, as otherwise they would be invisible. And since the functionality was already implemented, it was also added as a standalone component.

Particle generator

Often in games we want to create particle effects. Particle effects are created by simulating a large number of points and then rendering them. For the points to be stored efficiently, we only store the center point of each particle. When we later want to render the particles, we use the geometry shader to turn each particle into a quad that faces the camera.

The particles are stored in a cyclic array of a constant size, which is then mirrored as a vertex buffer on the GPU. If we were to update these positions every frame, we would perform a lot of API calls to DirectX which would cost us a lot of performance. For that reason, the particles remember their starting position and their velocity and only get sent to the GPU once. When rendering, the current position of each particle is calculated from the original position, velocity, and the time that passed since its creation.

Lastly, we would like the appearance of our particles to be customizable. For this purpose there is an asset class *ParticleMaterial* which remembers the pixel shader together with 2 textures.

Skybox

Pretty much all games use some sort of skybox 19. Skybox renders a constant image around the player giving the illusion of some sky or terrain infinitely far away. Looking back at the figure 2.7, we can see a skybox of a space being rendered.

Point light

The most common type of light in games is a point light [20]. A point light originates from a single point and falls off with distance. Typically, the fall of is modeled using three factors: *ConstantFactor, LinearFactor,* and *QuadraticFactor*. The intensity given distance *d* is then calculated using the formula in the figure 2.8.

$$(ConstantFactor + LinearFactor \cdot d + QuadraticFactor \cdot d^2)^{-1}$$

Figure 2.8: Formula for light intensity

To render our lights efficiently and minimize the amount of pixels that need to be calculated inside the pixel shader, we render our light only in the approximate sphere it affects. For lights that are far away, this minimizes the number of pixels rendered from a ~2,000,000 pixels (in full HD) to several thousands or sometimes even hundreds.

The rendering model used in our game engine is the phong reflection model [21] because of its simplicity.

Ambient light

Lastly, in almost all scenes there will be some ambient light. A light that is spread throughout the whole scene with no particular source. Without an ambient light, objects not directly facing the light source would be pitch black, which is unrealistic.

2.7. User Interface

User interface systems are typically implemented by either using a library that renders the interface using the DirectX, or by creating a custom user interface library from scratch using the DirectX. However, since the game engine connects directly to a WPF window, we can use the WPF itself to create a user interface system. The advantage of this is that the user is likely to be already familiar with the WPF. The WPF is also really powerful and allows us to create any user interface we might want.

The implementation of this is done by having a WPF *ContentPresenter* [22] on top of the window and allowing us to place a WPF *Grid* [23] containing the user interface on top of it.

2.8. Physics

To implement a physical system, we can choose between several libraries that perform real time physical simulations. The libraries we have available are Bullet Physics [24], PhysX [25] and Havok [26].

Havok's licence prevents it from being used for commercial use. This could cause issues for the users of our game engine so we decided to not use it. Between Bullet Physics and PhysX, the main difference is that Bullet Physics runs on the CPU while PhysX runs on the GPU. Not everyone has a good GPU and if we used PhysX, it could cause problems for people who rely on an integrated GPU. Performance wise, Bullet Physics seems to be comparable to PhysX. On top of that, Bullet Physics is open source, so we will use Bullet Physics for our game engine.

The Bullet Physics contains a class *World* which holds all of the rigid bodies and simulates their interaction over time. We will create a class *PhysicalSimulation* which will roughly correspond to this world.

Then we will provide a *RigidBodyComponent* corresponding to the rigid body from Bullet Physics. Our component will also remember the key properties of the underlying Bullet Physics rigid body to allow us to set it through the editor. Lastly, our component will also provide an interface to access the underlying rigid body to make the physics extensible.

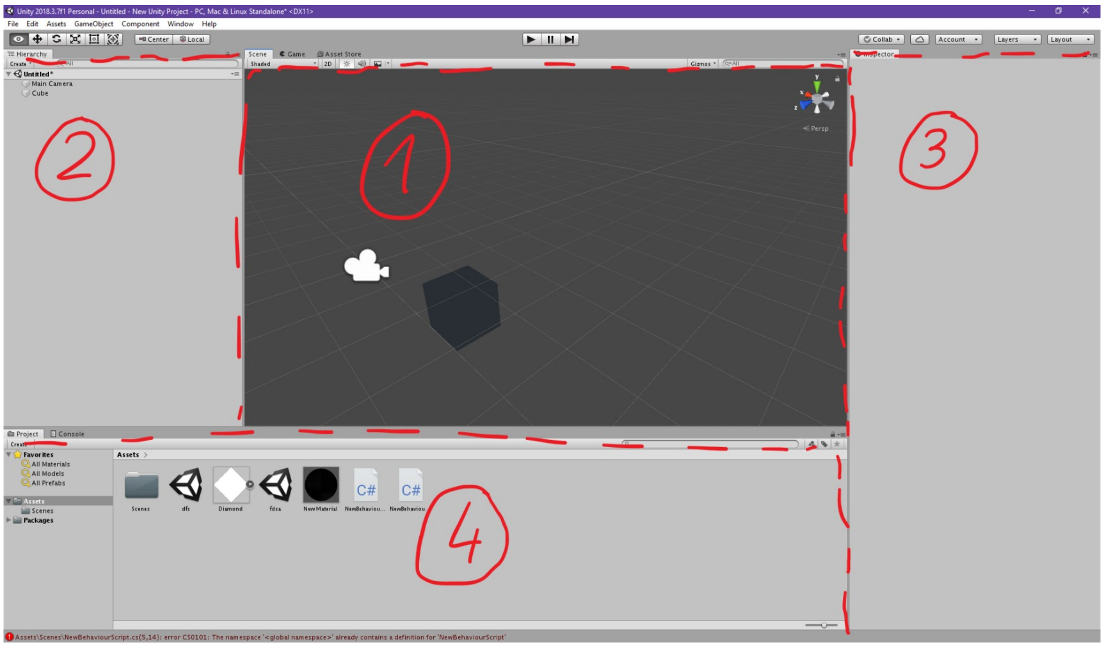
Together with *RigidBodyComponent*, our basic physics system will provide a *BoxCollider*, *SphereCollider*, and a way to create more colliders by extending the class *ColliderComponent* and implementing a method that returns a shape from Bullet Physics.

2.9. Editor

We need to decide on how our editor will look. In particular – what panels should it have?
We will certainly need:

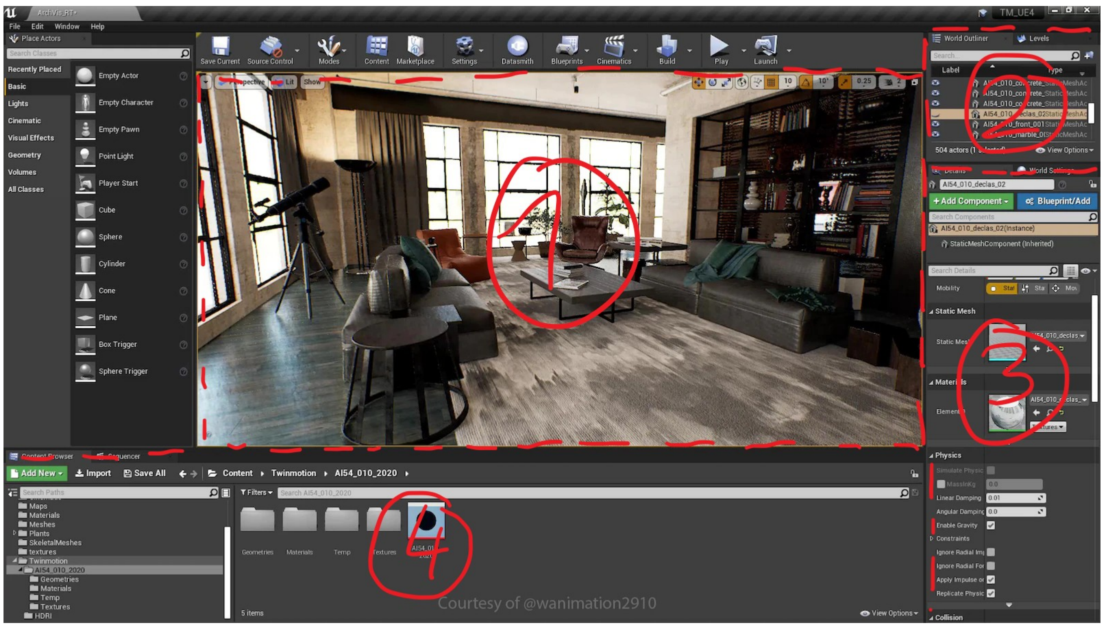
1. A central window that displays the world
2. A scene explorer – a panel with a tree structure of game objects inside the scene
3. An inspector – a panel for editing the currently selected game object and its components.
4. An asset browser – a panel that shows a file hierarchy with our resources so that we can assign them to our objects.

Looking at the popular game engines such as Unity [27] and Unreal Engine [28], we can see that they also have similar panels. Figure 2.9 shows a screenshot of the Unity editor and figure 2.10 shows a screenshot of the Unreal Engine editor. Lastly we can see a screenshot of our editor with the same panels (figure 2.11).

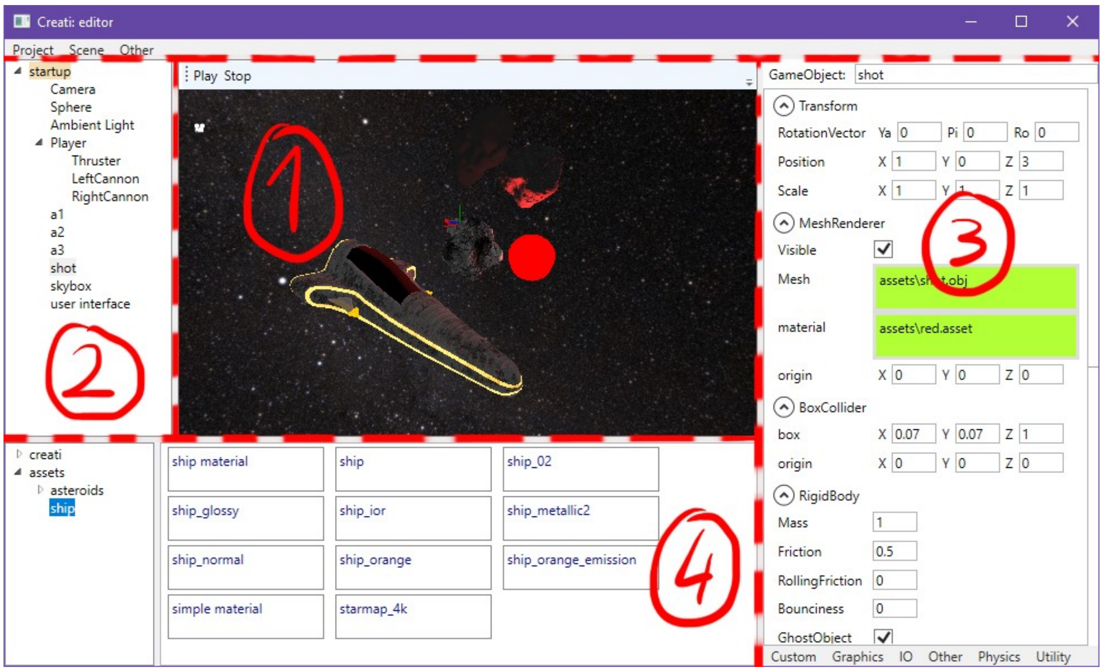


```
Assets\Scenes\NewBehaviourScript.cs(5,14): error CS0101: The namespace '<global namespace>' already contains a definition for 'NewBehaviourScript'
```

Figure 2.9: Unity user interafce



2.10: Unreal Engine user interface



GameObject: shot

Transform
RotationVector Ya 0 Pi 0 Ro 0
Position X 1 Y 0 Z 3
Scale X 1 Y 1 Z 1

MeshRenderer
Visible ✓
Mesh assets\shot.obj
material assets\red.asset
origin X 0 Y 0 Z 0

BoxCollider
box X 0.07 Y 0.07 Z 1
origin X 0 Y 0 Z 0

RigidBody
Mass 1
Friction 0.5
RollingFriction 0
Bounciness 0
GhostObject ✓

Figure 2.11: Creati Editor user interface

2.10. Assets

All games and game engines have to handle assets somehow. Assets are things like textures, shaders, etc. We need to be able to load files into classes and we need to ensure that each file gets only loaded once, otherwise we would waste a lot of space. Advanced caches can also automatically load objects beforehand and unload them when no longer necessary. This is however not needed in the game engine core yet as there are not enough features to expect games created in it to take up gigabytes of space.

To work with assets, we will use a single central class *Assets*. The class *Assets* will provide an interface to look at our files as if they were instances of classes. Since we don’t want to lock a certain resource type (or file extension) into only one possible type, we will allow the resources to be loaded into multiple types at once, and if possible, have the share some data with each other. Logically, we then need to index our resources by both the url and the class type.

Logically, we need a class that can load certain resource types into a specific class type. For example, we could have a resource loader that loads all the image types (.png, .jpg, etc.) to some *Texture* class. For this, there is the class *AssetLoader<T>*. Since one asset loader will always load into one class type, it makes sense to make it a generic class. This class will get a url of the file, load it into an instance of the class, and return that instance.

Our *Assets* class will need to know of all the *AssetLoaders* to be able to use them for loading. To avoid coupling and to make this process cleaner and easier for the user, the *Assets* class uses reflection to locate all the *AssetLoaders* in all the assemblies.

In most games, when an asset is not loaded, it uses some default value. When a model is not loaded, it shows a 3D text saying “ERROR” (figure 2.12), when a texture is not loaded, a checkerboard of white and pink is used instead. This is done so the developer can easily notice that something is wrong. If instead the mesh didn't get rendered at all, the developer might not notice that something is wrong.

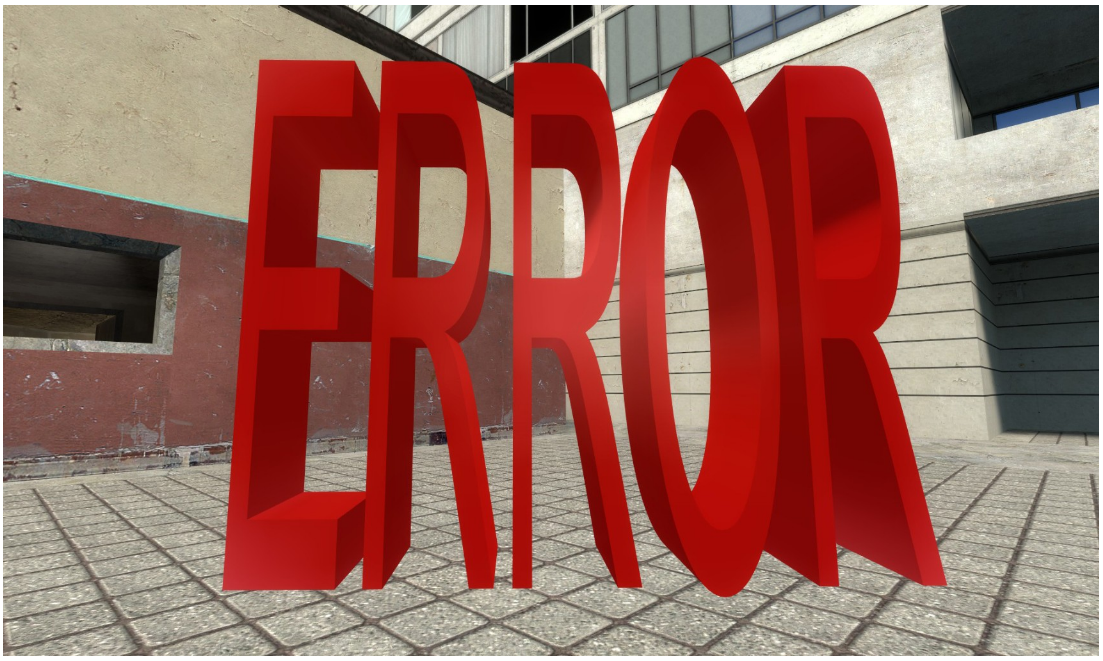


Figure 2.12: Error text being displayed after loading of a model failed

To set the default value, we will use a similar approach as with the *AssetLoaders*. The game engine contains a class *AssetDefault<T>*. By inheriting that class, we can define the default value for any class type. This class is then also collected using reflection.
Lastly, we would like to make it easier for the user to make custom assets. For example, let's say we have some *material* class. It would make sense to treat this class as an asset. There are several reasons to treat it as an asset. Assets can be more easily shared between different game objects and no game object is their "owner". We can also share the material class between different projects this way.

Having any class be an asset only makes sense if we can create instances of this ass and edit them inside the editor. To do that, we need to let the editor know that a particular class should be treated that way. In the game engine this is done by using the attribute *[AssetClass]*.

Resources

Which resource types should the game engine support by default? Resources used by the systems that are currently implemented. Most of those resources are thus related to graphics.

First of all, we need to load shaders (.hlsl). There is an *AssetLoader* for each shader type – vertex shader, geometry shader, hull shader, domain shader, and pixel shader. Vertex shader and geometry shader could also benefit from having an analysis performed on them using DirectX reflection [29]. The analysis performed is about the input layout of the shaders, because by knowing the input layout, we can automatically connect shaders with our meshes and simplify working with shaders for the user. The analysis results are kept inside a class called *MeshVertexShader.* The shaders are compiled using *D3DCompiler* [30].

Second type of resource we need are textures. Logically, textures are loaded as *Texture2D* [31]. However, to assign the texture to the GPU, we need to put it inside a *ShaderResourceView* [32]. Here we use a trick by letting the *AssetLoader* for *ShaderResourceView* use the *Texture2D* from the *Texture2D AssetLoader*. The textures are loaded using *Windows Imaging Component (WIC)* [33].

Lastly we need to load meshes. But first, we need to decide how we will even structure our mesh. Mesh is a list of vertices where each vertex can have various properties such as position, normal, tangent, bitangent, color, uv, and more. However, most shaders don't need all of this information; each shader needs a different subset. For that reason, all the information is split between multiple arrays. E.g. one array for position, one array for normals, one array for colors, and so on. After we have our *Mesh* loaded, we still can't use it. That is because all the information is on the CPU and not on the GPU. For that reason, each *Mesh* also holds a reference to *MeshDx*, which mirrors the same information but in buffers on the GPU.

The meshes are loaded using *AssimpNET* [34]. *AssimpNET* is a library that can load tens of 3D model formats. It is also the only good library that is usable in C#.

2.11. Serialization

Our game engine needs to be able to serialize various files. We need to serialize our scenes with all the game objects and components so we can save our project and then load it later, we can also use this feature to save the current state of the game. We also need to be able to serialize all the asset classes mentioned at the end of chapter *2.9. Assets*.

First, we need to decide on a file format to serialize into. The options we have are the following:

1. A binary format
2. XML [35]
3. JSON [36]

By using the binary format (1), our files would have the smallest size. However, a binary format is not human readable and so it makes the development more difficult. Furthermore, most of the data is efficiently stored in the actual resource files so using a larger format won't increase the size that much.

The choice is between XML (2) and JSON (3) [37]. The advantage of XML is that the structure of XML supports more features, such as comments, attributes, and more data types[1]. On the other hand, JSON files are smaller, faster to parse, and easier to read by humans. Since the extra features are not necessary, the game engine uses JSON as its format of choice.

The options for libraries for JSON serialization and deserialization in C# are pretty much only Json.NET [38] and the new namespace integrated inside the standard library *System.Text.Json* [39]. The problem is that neither of the serializers supports cycles. Furthermore, we want to use custom rules and attributes to decide which members get serialized. We also want the serializer to recognize assets and serialize them only as a relative url of the file it originates from. Because of the problems mentioned, the game engine uses a custom built serializer.

In order to make our asset classes equivalent to the regular assets, it makes sense that the serialization is accessed through the *Assets* class. In particular, the loading of regular assets and asset classes are both accessed through the same method.

For serialization it's only natural to use reflection.

---

[1]The JSON data types are limited to booleans, strings, and numbers.

Serialization rules

We need to decide which fields and properties should get serialized and which should not. Usually, fields hold the actual data while properties act as accessors to these data, so we will only serialize fields. Furthermore, private fields often contain some internal state that has nothing to do with the properties of the class itself, for that reason we will only serialize public fields by default but allow the user to mark private fields with the attribute *[SerializeField]* to force the serializer to serialize that field.

We will need to format following types of data:

1. Value types
2. Structs
3. Arrays
4. Class instances that are being serialized for the first time
5. Class instances that were already serialized
6. Class instances that originated from assets

Value types (1) like string, int, long, and bool, will be serialized directly as their respective value.

Structs (2) will get recursively serialized using the same process.

Arrays (3) will get serialized as regular JSON arrays.

When we are serializing the instance of a class for the first time (4), we will serialize it recursively the same way we would serialize structs. However, we also need to include some additional data. Since a field of type *A* could store any subtype of class *A*, we have to add a tag remembering that class type. This tag looks like *".type": "Creati.GameObject"*. Second, we need to remember some id for the class instance so we can refer to it from the consequent occurrences. The id will be an int and look like this: *".id" : 5*.

Repeated occurrences of class instances (5) will be serialized as a reference to the first occurrence. If the id of the first occurrence is 5, the serialized reference will look like this: *".ref": 5*.

Lastly we serialize assets (6). When a class originates from an asset, we don't want to serialize the whole asset again but we just want to serialize a url of that asset. So instead that whole class, we will just serialize a tag holding that url, which will look like this: *.asset": "creati\\models\\cube2x2x2.obj"*.

Some classes need special care after they get deserialized. To allow classes to "fix themselves" after deserializing, the game engine contains the interface *IDeserializable* with a method that gets called after deserializing.

In figure 2.13 we can see an example of a serialized scene.

```json
{
  ".id": 0,
  ".type": "Creati.Scene",
  "gameObjects": [
    {
      ".id": 1,
      ".type": "Creati.GameObject",
      "Scene": {
        ".ref": 0
      },
      "components": [
        {
          ".id": 2,
          ".type": "Creati.Transform",
          "Position": {
            "X": 7,
            "Y": 5,
            "Z": 29
          },
          "Scale": {
            "X": 1,
            "Y": 1,
            "Z": 1
          },
          "Rotation": {
            "X": 0.08248053,
            "Y": 0,
            "Z": 0,
            "W": 0.9965927
          },
          "rotationVector": {
            "Ya": 0,
            "Pi": 0,
            "Ro": 0
          },
          "gameObject": {
            ".ref": 1
          }
        },
        ...
}
```

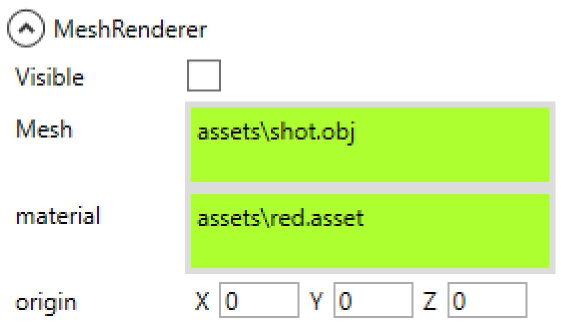
Figure 2.13: Example of a scene serialized by the custom serializer

2.12. Inspecting

The integral part of the editor is the inspector. The inspector needs to be able to display and allow us to edit the properties of all components of the selected game object. To do that, we will naturally use reflection. We will iterate over all the components, generate a user interface for each component separately, and then add them all up together.

Now the problem is reduced to generating a user interface for each component. We would like to be able to generate the user interface without the user having to do anything, but we would also like to be able to generate a completely custom user interface. To do that, we simply allow the user to override the method *Inspect(...)*, which generates the user interface. The user interface generated is in the form of a *FrameworkElement* [40].

However, most of the time, we would like the editor to generate the user interface. For example, take a *MeshRenderer*, a component class that renders meshes. This class contains 4 fields: boolean *visible*, *mesh*, *material*, and *origin* (vector). What we want is to generate an editor for each of those properties and join them together. We would like to generate a checkbox for the boolean value *visible*, some asset selectors for the *mesh* and *material*, and then some numerical editors for the *origin*, like we can see in the figure 2.14.



MeshRenderer
Visible ☐
Mesh assets\shot.obj
material assets\red.asset
origin X 0 Y 0 Z 0

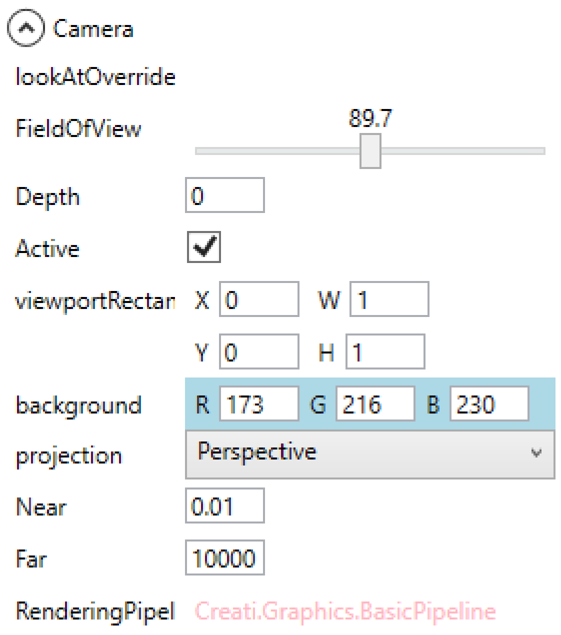
Figure 2.14: Editor for a MeshRenderer

The editor that we generate clearly depends on the type of the property. Booleans will always want a checkbox, numbers will almost always want a numerical textbox, and structures will most likely want to be recursively deconstructed into editors. However, we might sometimes want to generate a different editor for the property without writing the whole interface code. For example, sometimes it makes more sense for the numerical value to be edited using a slider, like we see in the figure 2.16.

The obvious solution for this is to use attributes. Using attributes will be very intuitive for the user. If we define the whole editor class as an attribute, we can even use the whole editor as the attribute for defining the editor, like in the figure 2.15.

```
[FloatSliderEditor(Min = 1, Max = 179, DecimalPlaces = 1)]
public float FieldOfView;
```

Figure 2.15: Specifying the editor to be used inside the attribute



Camera
lookAtOverride
FieldOfView 89.7
Depth 0
Active ✓
viewportRectan X 0 W 1
Y 0 H 1
background R 173 G 216 B 230
projection Perspective
Near 0.01
Far 10000
RenderingPipel Creati.Graphics.BasicPipeline

Figure 2.16: FieldOfView using a slider instead of a textbox

Editor

In particular, there is a class *Editor*, which can generate a user interface if we give it the information necessary for accessing the property. As mentioned before, this class inherits the class *Attribute* to allow us to assign particular instances of the *Editor* subclasses to various properties. Since often we want the *Editor* to be used for all properties of a certain type, we can also mark the *Editor* class the attribute *[DefaultFor(type)]*.

We will also need the editor to provide a default *Editor* for certain types. Logically, we want pretty much all of the primitive types to have a default editor, since these types will be used very often in user defined components. These types are: all numerical types (*int, long, etc.*), *string*, *bool*, and all *enum* subtypes. There are also certain structs that will be used commonly, most of them belonging to the library *SharpDX*. These types are mostly vectors and *Color*. The user might also want to link game objects together by having some components hold a reference to a different game object. Thus we will also need a game object editor. Since we don't want to create those game objects, we will just have the user drag and drop them from the game object hierarchy in the editor.

Lastly, we need to handle resources somehow. Assuming we know how to load a resource, we can borrow the idea from the game object editor and also make it drag and drop. The problem that arises with this is that all classes can potentially be resources. There is no telling if the class can be loaded from some file. On top of that, the classes that can be loaded might change if the user adds an *AssetLoader* for a new class type. This is solved by having the editor generate an editor for each type that can currently be loaded.

Not all members should have an editor for them. By default, we only want to edit members that will get serialized, i.e. public fields and private fields with *[SerializeField]* attribute. Editing members whose information will inevitably get lost would not make any sense. However, we might want to generate an editor for properties for example because their underlying field will get serialized. The game engine provides attributes *[ShowEditor]* and *[HideEditor]* to override the default behavior. Furthermore, if a member has an attribute of type *ValueEditor* (like in the figure 2.16), we will also generate an editor.

Whenever we change a value of a class using the editor, we need to notify the class. For example if we change the material of the *MeshRenderer*, the *MeshRenderer* needs to update the rendering graph. Sometimes we might also want to reset the component to its default value. To do that, we allow all classes to implement the interface *IInspectable* which has methods *Validate()* and *Reset()*. *Validate()* is called every time the editor changes any value. *Reset()* is called everytime we want to reset the class to the default value. The *IInspectable* interface also contains the method *Inspect(...)* allowing the user to override the user interface generated for that class.

Besides components, we also want to be able to inspect the asset classes (classes with the attribute *AssetClass*). The process of generating the editor is the same as for game objects and components.

Lastly, it would also be nice if we could call methods on the classes from the editor. Calling methods with parameters wouldn't be impossible but it would be difficult to implement. Since this feature is not crucial, we only implemented calling of parameterless methods. Whenever we mark a parameterless method with *[ShowEditor]*, a button which calls that method gets generated inside the editor (figure 2.17).

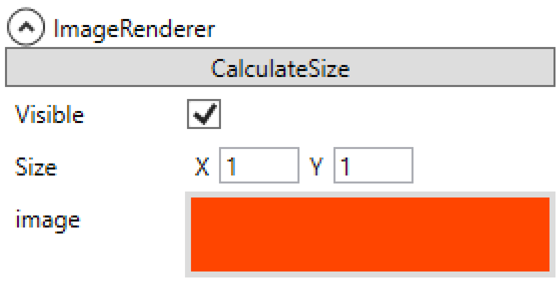

ImageRenderer
CalculateSize
Visible ✓
Size X 1 Y 1
image

Figure 2.17: An editor for a method being generated as a button

ValueContext

One question we didn't answer is how do we connect editors with the various fields and properties inside classes?

The game engine solves this by using a class *ValueContext*. This class serves as an interface to manipulate with a value inside some field or property. Internally, this class contains a getter and setter for the particular member of the particular instance of a class. *ValueContexts* for a class create a tree mirroring the hierarchical structure of the class. This tree is then used to distribute events between the root and the leaves.

The game engine solves this by using a class *ValueContext*. This class serves as an interface to manipulate with a value inside some field or property. To manipulate the value of some class we need two things. First is an event (*UpdateVisual*) that occurs every time the value in the corresponding class gets changed and the visual element needs to be updated to match it. Second we need a method (*ValueChanged(...)*) to update the value inside the class after the corresponding visual element changes.

Internally, this is done by remembering a *setter* and a *getter* for the member, together with the instance of the underlying class. These *ValueContexts* are then arranged in a tree mirroring the hierarchical structure of the class we are editing, with the instance of that class at the root. This tree structure is needed to distribute events about changes between the leaves and the root. When we change a value in the editor, we need to bubble an event saying that we need to call the *Validate()* method. Figuring out whether the underlying class was changed or not is complicated and unnecessary, we can simply bubble down an event every frame saying to update the visual representation to match the new values.

2.13. Scripting

Any serious game engine needs a way for the developer to create their own components or even modify the game engine itself. To do this, when a project in the editor is created, a new visual project is generated together with it.

The idea is that the user will program their own components and then build the project. Afterwards, the game engine reloads[1] the game assembly and includes all the new components and behaviors created by the user. It would be unacceptable for the user to have to reload the whole editor to load the scripts.

Because after reloading the assembly, the components are different internal types and might not have matching fields, we need to somehow transfer the old components with old fields into new components with new fields. This is solved by having the game engine save the scene in a temporary file, unload the scene, reload the assemblies, and then load the scene again from the temporary file. Loading the scene after some of the component's fields changed causes it to only load the fields that have the same name. New fields will not get loaded and old fields that no longer exist will get discarded. This is nonetheless the desired behavior.

---

[1]Reloading the assembly is done through the menu option *Other → Reload*.

We need a way to load and unload dlls without restarting the game engine. There exists a class *AssemblyLoadContext* [41] which provides the behavior we want. We just create an *AssemblyLoadContext* for our dll and use *Load()* and *Unload()* to load and unload it.

By loading the dll, we are keeping the file open. This would prevent us from building the game project and updating it. To avoid this we simply copy the dll file and everything else necessary to a temporary directory.

2.14. Exporting

The last step of any game development process is to export the game so it can be run outside of the game engine. Since the game engine already knows how to load external dlls and run the game, we can create a launcher that does the exact same thing, just without the editor.

Using this idea, the game engine exports the game by creating a new output directory, copying the launcher with all the dlls there, and then putting the game dll on an expected place. The launcher will then simply open the game dll from the expected location and start a scene. The scene that will be started is always a scene called *startup.scene*.

3. Developer Documentation

The thesis comes in two visual studio solutions:

- *Creati* – containing the actual game engine
- *game* – containing the demo

The architecture is depicted in the figure 3.1.

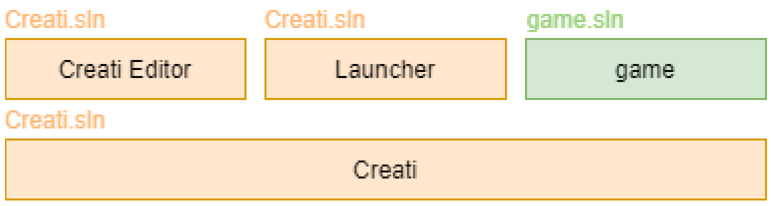


Figure 3.1: Project dependencies

Creati is the game engine core. All other projects reference the project *Creati*. The output of this project is a dll file *Creati.dll*.

Creati Editor actual editor application. The editor is written in *WPF*. The output of this project is the main output of the thesis – the executable *Creati Editor.exe*.

Launcher is a small project for the game launcher. It tries to load the game from the relative path *game/game.dll* and launch the scene *game/assets/startup.scene*.

Game is a demo game made using the *Creati Editor* to create a tutorial for using the game engine and editor.

Licence

All of the projects in this thesis are provided under the MIT licence. More details can be found in LICENCE.TXT in the root directory of the attached CD.

3.1. Compilation

When compiling the *Creati Editor* project, directories *project template* and *export template* get copied in the same directory.

Project template

The directory *project template* has a file structure from figure 3.1.

```
assets/
    (assets and scenes for our game)
creati/
    (assets provided by the game engine and needed by the editor)
references/
    Creati.dll
    (other dlls referenced by the project)
ConstantForce.cs
game.csproj
game.sln
```

Figure 3.2: Project template file structure

Export template

The directory *export template* has a file structure from figure 3.3.

```
runtimes/
    (more dlls)
Creati.dll
Launcher.dll
Launcher.exe
(other necessary dlls)
```

Figure 3.3: Export template file structure

3.2. Creati projects

Creating and loading projects

When creating a project, two things are done. First the *project template* directory gets copied to the target location. Then a file called with the extension of *.creatiproject* is created inside the project directory. This file is completely empty.

When opening a project, we show a dialog for selecting a project file with the *.creatiproject* extension. This file only serves as an "anchor" to set the working directory for the project and locate the *assets/startup.scene* and *bin/.../game.dll* relative to that directory.

Creating and loading projects is done using the commands [42] *NewProject* and *OpenProject*.

Updating projects

When the game engine project gets updated, the projects don't see it because they are still using the outdated dlls, i.e. *Creati.dll*. For that reason, whenever a project is being loaded in the editor, it performs an update. The update is done using the command *Update*, which copies all the *.dll* and *.xml* files from the editor directory to the creati project directory. After that, it also copies all of the new assets inside the *creati* directory.

Exporting projects

When a creati project is being exported, the editor first clears the *output* directory and then copies the contents of the *export template* (figure 3.2) inside of it. Afterwards, the directories *assets* and *creati* from the project directory get copied inside the *output* directory.

We can see the resulting file structure in figure 3.4. The *Launcher.exe* always expects the *startup.scene* and *game.dll* to be at these exact locations.

```
Launcher.exe
(dlls)
game/
   assets/
      startup.scene
      (other assets)
   creati/
      (engine assets)
   game.dll
```

Figure 3.4: Folder structure of the exported game

3.3. Editor

Architecture

The editor follows the MVVM pattern [42]. The main editor window class is the *EditorWindow*. This *EditorWindow* uses a singleton *EditorViewModel* as its *DataContext* [43]. The individual panels of the *EditorWindow*, with the exception of the world renderer, are then stored in their own respective files: *AssetExplorer.xaml, Inspector.xaml,* and *SceneHierarchy.xaml*.

Most interactions with the *EditorViewModel* are done through commands [44]. The class *EditorViewModel* contains a static property for each command it supports. The only exception to this is the method *Init(projectDirectory : string)*; this method is called to load the creati project after we have selected the project directory.

The *EditorViewModel* also holds the current logical state of the editor. That means the reference to the currently edited scene, selected game object, opened asset directory and the inspected object.

Editor loop

Instead of using the game loop, the editor uses its own custom loop and only switches to the actual game loop when debugging. This loop is stored in the class *EditorLoop*. This loop manages the WPF window and renders the special *EditorCamera*.

Debugging

To enter the debug mode, the editor first saves all the changes that were made in our files. The currently opened scene is serialized as *temp.scene* and the scene object is stored away in a temporary variable. The editor then loads the *temp.scene* it just saved and sets it as the active scene. Then the game engine's loop is run. When the debugging ends, the scene used for debugging is discarded and the original scene object we stored away is used again.

The game engine's loop that is run is modified to include the *EditorViewModel.Update()* method, which keeps the inspector updated.

Editor camera

The editor uses a special camera which exists outside of the scene and uses the *EditorPipeline* for rendering. This *EditorPipeline* extends the *BasicPipeline* by also rendering the *EditorRendering* class.

When rendering the *EditorRendering* class using the *Render(...)*, we have to give it a reference to the currently inspected game object as well as the component that our mouse is hovering over. This information is then used to give users the functionality of conditional rendering by calling *EditorRendering.Selected(...)* and *EditorRendering.Hovered(...)*.

3.4. Rendering

Rendering

On top of all rendering is the static class *Rendering*. This class has a static method *RenderAllCameras()* which is called every frame at the end of the loop.

The class *Rendering* additionally provides two utility methods:

- *RenderCamera(camera : ICamera)* – renders a particular camera over the whole screen. This is used in the *EditorLoop* to circumvent the usual rendering process.
- *RenderCameraPreview(camera : ICamera, scale : float)* – renders a camera preview scaled down at the bottom right corner. This is called in the *EditorLoop* whenever we select a game object with a camera.

The rendering process for each camera is the following:

1. Set static variables related to the camera
2. Generate the viewport
3. Update the constant buffer for camera
4. Call *Clear(...)* and *Render(...)* on the camera's rendering pipeline

In the first step (1) we set some static properties inside the static class *Rendering* so they can be later accessed in the rendering process. These properties are:

- *Viewport : Viewport*
- *View : Matrix*
- *Projection : Matrix*
- *CurrentlyRenderedCamera : ICamera*

The viewport (2) inside the camera is in normalized units (0,0) to (1,1) and we need to convert it to pixels.

The *ConstantBuffers.Camera* (3) which is used in various shaders for light rendering needs to be updated with each camera.

The last step (4) is to clear the viewport so it's ready for rendering and then to render it using whichever rendering pipeline the camera uses.

Rendering pipeline

Interface *IRenderingPipeline* defines two methods that get called during the rendering:

1. *Clear(viewport : Viewport)*
2. *Render(viewport : Viewport, background : Color)*

The clear method (1) should clear the part of the screen defined by the viewport. We don't want to clear the whole screen because if we are rendering multiple cameras at the same time, clearing the whole screen would discard the rendering results of the previous cameras.

The render method (2) controls the rendering process. The actual rendering is split between static classes *SurfaceRendering, LightRendering, ForwardRendering* and *EditorRendering*. This method simply calls the *Render(...)* method on each of them in the correct order and with the correct parameters. Each of these rendering classes contains a *RenderingGraph* containing all the graphical components that need to be rendered.

Rendering settings

Here we will discuss the special settings that each of these rendering classes needs to set.

- *SurfaceRendering* sets a list of textures (G-Buffer) as its rendering output.
- *LightRendering* sets the output textures from the *SurfaceRendering* as its input and also changes the *BlendState* to additive so that all the light contributions get added together.
- *ForwardRendering* and *EditorRendering* sets a blend state that enables transparency through the use of alpha.

Rendering graph

Rendering is done through the use of so called *RenderingGraph*. Objects that the *RenderingGraph* can render are a collection of rendering steps. Each *RenderingStep* can be entered through the method *Draw(context : DeviceContext)* and exited through the method *Exit(context : DeviceContext)*.

RenderingGraph holds these individual graphical objects in a tree structure where each node corresponds to a *RenderingStep*. Rendering of the rendering graph is done recursively. When we are rendering a node, we first call the *Draw(...)* method, then we iterate through all the children and recursively draw them, and afterwards we call the *Exit(...)* method on the rendering step associated with the node.

A node of the rendering graph is under the class *StateLayer*. The *StateLayer* actually keeps the leaf nodes and the children nodes separated, and whenever it enters a leaf node, it checks whether the underlying *RenderingStep* is dead. A rendering step dies when the corresponding component gets destroyed.

To recognize whether the corresponding component is destroyed, the rendering step *DrawMethod* must be used as the last rendering step. When we initialize a *DrawMethod* with a method inside of our component, the *DrawMethod* extracts the underlying component from the method and then uses it to determine whether the rendering step is dead.

DirectX classes

This chapter discusses the lower level DirectX classes used throughout the game engine.

Most DirectX classes are kept inside a static class called *Devices*. This class contains *Direct3D11Device, Direct3D11DeviceContext, Direct2DDevice, Direct2DDeviceContext, Direct2DFactory, DirectWriteFactory*, and *WICImagingFactory*. All of these are initialized inside the static constructor.

There are also several static classes that hold commonly used DirectX states of the type in the class name. These are again initialized inside the static constructor.

- *BlendStates*
- *DepthStencilStates*
- *RasterizerStates*
- *SamplerStates*

There exist two utility classes for loading DirectX resources: *TextureLoader* and *ShaderCompiler*. These classes are then used directly by the respective asset loaders.

Lastly, there is a class *RenderingTexture*, which couples together *Texture2D* and the views on that texture, such as *RenderTargetView, ShaderResourceView,* and *DepthStencilView*. This class allows us to resize all of the views at once using the method *ResetTexture(newTexture : Texture2D)*. It is used commonly throughout the renderers.

Shader analysis

To analyze the shaders, the game engine uses the class *ShaderAnalysis*. The class *ShaderAnalysis* uses DirectX reflection to figure out useful information about shaders. The information provided by the shader analysis is:

1. Indices of textures, constant buffers, and samplers
2. Input semantics
3. Formats of input elements
4. Primitive topology (for geometry shaders)

This shader analysis is then used inside *MeshVertexShader, VertexShaderView,* and *GeometryShaderView*.

3.5. WPF Window

In this chapter we will talk about the two classes that allow us to connect the DirectX with the WPF

WPF

Class WPF contains a single static method *ProcessMessages()*. This method processes the messages that windows is sending to the WPF. By calling this method periodically, we can have our own loop that controls the WPF, not the other way around.

The *WPF.ProcessMessages()* is called periodically inside the game loop.

Game surface

Class *GameSurface* abstracts the idea of the WPF window and connects selected *ContentPresenter* with the DirectX rendering output.

To use the *GameSurface* class, we need to initialize it by calling the static method *Initialize(window : Window, target : ContentPresenter)*. This method initializes a directx render target from *Microsoft.WPF.Interop.DirectX* namespace inside the *ContentPresenter*.

On top of that, it also adds two grids to it. The *UIGrid* and the *UIPreviewGrid*. The *UIGrid* is used by our user interface system to display the user interface. The *UIPreviewGrid* is used to show a preview of the user interface in the editor.

The *GameSurface* class manages presenting the render output inside our *ContentPresenter* and also holds a *Texture2D* of the backbuffer we will be rendering into. It also facilitates all of the communication with the WPF – keyboard events, mouse events, querying the window size, and resize event.

3.6. Input

VirtualKey

The enum *VirtualKey* contains key codes for all the keyboard keys, mouse buttons, and even mouse wheel.

Input

All of the input management is done inside class *Input*. All keys are hidden behind a string name.

When a new key binding is registered using the method *SetBinding(name : string, key : VirtualKey)*, it creates a class of type *KeyBinding* and inserts it to an internal dictionary with the key of *name*.

Each *KeyBinding* remembers whether it’s up, down, or pressed.This information is updated every frame. A key is pressed if it is down but was up the previous frame. The information about the state of the key is updated by querying the static classes *Keyboard* and *Mouse*. These classes belong to the namespace *System.Windows.Input*.

3.7. User Interface

To display the user interface, the game engine uses the static class *GameSurface* which contains a grid over the rendering target. We can add elements to this grid using the property *GameSurface.UIElements : UIElementCollection*.

To create custom interfaces, the users inherit the class *GridComponent*. The *GridComponent* works by adding itself into the aforementioned collection inside the *Start()* method and removing itself from it in the *Destroy()* method.

Because the *GridComponent* needs to inherit the class *Grid*, we can’t inherit the base *Component* class but we must instead implement the *IComponent* interface all over again.

3.8. Assemblies

Class *AssemblyManager* is used for two things: loading the game dll and enumerating types for reflection.

Game assembly

The property *AssemblyManager.GameDllPath* contains the path of the game dll used. This property is set at the beginning of the program runtime by either the *Creati Editor* or the *Launcher*.

Before loading the dll, the *AssemblyManager* always copies into a directory *temp/N*, where the *N* is the lowest available number. This is necessary because sometimes the assembly does not get unloaded successfully and then we cannot delete it.

AssemblyManager also provides a method that reloads the dll called *ReloadGameDevelopmentAssembly()*. It first unloads the current dll and tries to delete as many temporary files as possible. Then it creates a new directory *temp/N* and copies the dll file together with all the dependencies there. Lastly it loads the dll using the class *AssemblyLoadContext*.

Reflection

Whenever the game engine needs to use reflection, it uses one of the following enumerators on the class *AssemblyManager*:

- *RelevantAssemblies* – returns all relevant assemblies, which are the assembly of *Creati* and the assembly of the game dll
- *RelevantTypes* – returns all relevant types, which are all types of the relevant assemblies

Since the game assembly will get reloaded many times, it also provides the event *AssembliesChanged*, which is triggered every time the game assembly is reloaded

3.9. Inspecting

Value context

The core of the inspecting model is a class *ValueContext*. This class provides us with an interface to manipulate a particular field or property of an object.

Event *UpdateVisual* is called every time the editor wants us to update the visual element to match the visual components. This event gives us an object containing the new value of our field or property.

Method *ValueChanged(newValue : object)* should be called every time the user edits the value inside of the editor. Calling this method causes the underlying field or property to get updated to this value.

The *ValueContexts* create a tree structure mirroring the hierarchy of the object we are editing. The *ValueContext* in the root corresponds to the object itself. This tree allows us to bubble events down from root to the leaves and up from the leaves to the root.

The value context also contains additional information which can be used to create the editors:

- *Type : Type* – type of the underlying field or property
- *DisplayName : string* – name of the underlying field or property
- *Level : int* – the depth inside the *ValueContext* tree

Value editor

ValueEditor is a class used to generate *FrameworkElement* from a *ValueContext* to edit a field or property. Classes inheriting the *ValueEditor* must implement a single method: *EditValue(context : ValueContext)*, which returns the *FrameworkElement* for editing.

Editor lookup

To automatically find suitable value editors for the field or property, the editor uses the class *Editor*. The class *Editor* contains following methods:

- *EditContext(context : ValueContext)* – this method find the suitable *ValueEditor* for the particular *ValueContext*
- *EditMember(parentContext : ValueContext, memberName : string)* – this method finds a suitable *ValueEditor* for a member of the class or structure of the *parentContext* with the particular *memberName*.[1]
- *EditViableMembers(context : ValueContext)* – returns a list of editors, one for each member that should have an editor

[1]If we have a *ValueContext* that is used to access an instance of *Vector3*, we can call *EditMember(ValueContext, "X")* to get an editor that edits the *Vector3.X* value.

Editor selection works in the following way:

1. Check for the attribute of type *ValueEditor* and use that if possible
2. Check the dictionary of *Type → ValueEditor*. This dictionary is generated using reflection by looking up value editors with the *[DefaultFor(type)]* attribute and stored inside *EditorDatabase*.
3. If it's enum, use the enum editor
4. If it's observable collection, use the collection editor
5. If it's a method, generate a button
6. If it's a struct, recursively generate editor for each field and property

The *EditViableMembers(...)* will generate an editor for a member if:

1. The member does not have the *[HideEditor]* attribute and
2. The member has the *[ShowEditor]* attribute or
3. The member has a *ValueEditor* as attribute or
4. The member is a public field or
5. The member is a private field with the *[SerializeField]* attribute

IInspectable

Interface *IInspectable* has three methods:

- *Reset()* – this method is called when we reset the object inside the inspector
- *Validate()* – the method that is called every time a value is changed inside the editor which allows the objects to return themselves to a defined state
- *Inspect()* – returns the *FrameworkElement* for inspecting the class. By default, this method calls the *Editor.EditViableMembers(...)*.

Class *Component* implements this interface with virtual methods.

Inspector

Whenever the user clicks on a game object in the scene hierarchy, the editor needs to inspect it. To do that, it generates a textbox for the game object's name and then an inspector for each component.

The *Inspector* class holds a *ValueContext* and a *FrameworkElement* which define it. It then provides methods: *UpdateVisual(), TryValidate(),* and *TryReset()*, which will bubble down the *ValueContext* tree and call the appropriate methods.

EditorHelper

The method *EditorHelper.NameWrap(...)* wraps a *FrameworkElement* together with a label. This label is automatically determined using the *ValueContext*. When name wrapping the element, the *EditorHelper* checks the depth of the component. If the depth inside the *ValueContext* is at least 2, it inlines the editor next to the label. If the depth is 1, it creates padding.

3.10. Assets

All access to assets is done through the static class *Assets*. This class uses reflection to look up all the asset loaders and asset defaults. When loading a file, the *Assets* class first looks at the extension of the file. If the extension is either *.asset* or *.scene*, it uses the JSON deserializer. Otherwise it tries to find a relevant asset loader for that type and extension.

Class *Assets* provides access to the list of all types it can load using the method *AssetTypes()*. Furthermore, it registers a drag and drop editor for each supported *Type*.

Lastly, the class *Assets* provides method *GetAssetSource(asset : object)*, which checks if the object got loaded through the *Assets* class, and if it did, returns the original uri.

Serialization

Serialization is done through the class *Serializer*, which internally creates and uses instances of classes *JsonFileSerializer* and *JsonFileDeserializer*.

When *JsonFileSerializer* tries to serialize an object, it calls the *Assets.GetAssetSource(object)* to see if the object originated from an asset. If the object originated from an asset, it will serialize it as an asset and include only the url. If the object didn’t originate from an asset, it uses an internal dictionary to remember if they have already been serialized and under which id.

JsonFileDeserializer does the exact opposite of what *JsonFileSerializer* does. When deserializing an object that is said to be an asset, it uses the *Assets.Get<T>(url)* to load the object. For other objects, it remembers which id corresponds to which object so it can assign the same reference for repeated occurrences.

4. User documentation

4.1. Creating game objects programmatically

To add a new game object to the scene, we call the method:

```
scene.AddGameObject(name : string);
```

To add a component to the game object, we call the method:

```
gameObject.AddComponent<ComponentType>();
```

To destroy a game object, we call the method:

```
gameObject.Kill();
```

We can also destroy a particular component on the game object by calling

```
component.Destroy();
```

An example code for initializing the whole game object from code:

```
// Create a new game object in the scene
var go = scene.AddGameObject("Lightbulb");

// Set the transform component, using the property shortcut, to
the position we want
```

```
go.Transform.Position = new Vector3(0, 0, 5);
var transform = go.AddComponent<Transform>();

// Add an ambient light component, which makes it emit light, and
set its color
var ambient = go.AddComponent<AmbientLight>();
ambient.Color = new SharpDX.Color(0.1f, 0.4f, 0.3f);
```

4.2. Component lifetime

To create a new component, we create a new class inheriting the base class *Component*. This class must have a parameterless constructor. The constructor should be in almost all cases empty as the component is in an undefined state at that point. It doesn't even belong to the game object yet.

To initialize our component, we can override the method *Start()*.

```
protected override void Start() {...}
```

Any code that should be performed when the component gets destroyed, we override the method *Destroy()*.

```
protected override void Destroy() {...}
```

After a new game object is created, the method *Start()* is always called at the beginning of the game loop for all added components. That means that if we programmatically initialize a game object, the components will see each other when the *Start()* method gets called on them. However, the order in which the *Start()* method gets called is undefined. The *Destroy()* method gets called at the same point in the game loop for all the destroyed components.

To figure out whether a *GameObject* or *Component* is alive, we can use the property *Alive*.

4.3. Finding components

Each component holds a reference to its game object inside the property *GameObject*. We can use this property to access components inside the same component using the following methods:

```
// get all components of the type RigidBody
rigidBodies = GameObject.GetComponents<RigidBody>();

// get a component of type RigidBody if it exists, otherwise null
rigidBody = GameObject.TryGetComponent<RigidBody>();

// add a new component of type RigidBody and return it
rigidBody = GameObject.AddComponent<RigidBody>();

// return the first component of type RigidBody, and if it doesn't
```

```
exist, create it
rigidBody = GameObject.Component<RigidBody>();
```

The class *Component* also contains a shortcut method corresponding to the *GameObject.Component<RigidBody>()*:

```
rigidBody = C<RigidBody>();
```

The only exception is the *Transform* component which is always present and accessed through the property of *Component* and *GameObject* called *Transform*.

4.4. Finding other game objects

Sometimes we want to find other game objects on the same scene. We can access the scene of the game object using the property *Scene* on either *GameObject* or *Component*.

To enumerate over all game objects on the scene, we can use the property *GameObjects*:

```
foreach (var go in Scene.GameObjects) { ... }

// We can also enumerate over the whole Scene for the same effect
foreach (var go in Scene) { ... }
```

Because game objects exist in hierarchy, we can use the enumerator *GameObjectsUnpacked* to enumerate over all game objects, including children:

```
foreach (var go in Scene.GameObjectsUnpacked) { ... }
```

Lastly, when we want to find a game object with a particular name, we can access the Scene indexed by the name:

```
// This also finds game objects deeper in the hierarchy
// Only the first game object with that name is returned
var playerGo = Scene["Player"];
```

4.5. Scenes

The class *SceneManager* manages the active scene. It contains a single property *Active*, pointing to the active scene, and an event *ActiveChanged*, called every time the active scene gets changed.

To change the active scene, we simply change the value in the *SceneManager.Active* property. Typically we will load the scene from a file, resulting in following code:

```
// If we want to use the same instance of the scene
SceneManager.Active = Assets.Get<Scene>("assets\\scene2");

// If we want to reload the scene
SceneManager.Active = Assets.LoadAgain<Scene>("assets\\scene2");
```

4.6. Simple behavior inside components

The simplest way to create a behavior is to subscribe to one of the periodic event functions. Static class *Events* contains 3 events we can use.

1. *EarlyUpdate*
2. *Update*
3. *LateUpdate*

EarlyUpdate and *Update* are called in sequence, then the physical simulation gets updated, and then *LateUpdate* gets called. To subscribe the event, we first create our event handler method like this:

```
private voice Update() {...}
```

And then register it as the event handler for the *Update* event like this:

```
Events.Update += Update;
```

At some point, we might want to get the current time information about the scene. For this exists the class *Time*. Class *Time* contains a property *Total*, which tells us the total time of the game running, and a property *Scene, which* tells us the time of the current scene only. The time we get inside a struct called *GameTime*, which contains both the *Total* time elapsed and the *Delta* since the last scene.

We can can access these time properties in the following way:

```
Time.Scene.Total; // returns TimeSpan
Time.Scene.Delta; // returns TimeSpan
Time.Scene.TotalSecond;  // returns float
Time.Scene.DeltaSeconds; // returns float
```

4.7. Working with assets

The access to all assets goes through the static class *Assets*. This class has 3 methods for working with resources:

1. *Get<T>(url : string)*
2. *LoadAgain<T>(url : string)*
3. *Save<T>(object : T, url : string)*

When we use the *Get<T>(url : string)* method (1), the cache first checks if we have already loaded this object, and if we have, it returns it. If we haven't loaded this object yet, it tries to find an asset loader for that type. If even that fails, it returns the default value for that type.

LoadAgain<T>(url : string) (2) does the same thing as *Get<T>(url : string)*, except it doesn't look in the cache and always loads the file again.

Lastly, the method *Save<T>(object : T, url : string)* uses the JSON serializer to save the object we give it.

Serialized resources

All classes that have been serialized using the JSON serializer (by calling *Assets.Save<T>(...)*) have either the extension ".asset" or ".scene". All such files can in turn be loaded back using *Assets.Get<T>(...)* or *Assets.LoadAgain<T>(...)*.

Shaders

All DirectX shader classes can be loaded directly from the hlsl files into the corresponding shader classes (*PixelShader, VertexShader,* etc.).

Class *GeometryShaderView* stores the geometry shader (property *Shader*) plus the *PrimitiveTopology* discovered through the DirectX reflection.

Class *VertexShaderView* stores the vertex shader (property *Shader*) plus the original *ShaderBytecode* under the property *Bytecode*.

Class *MeshVertexShader* uses DirectX reflection to perform a thorough analysis of the input layout. It has properties:

- *Shader* – the vertex shader
- *InputLayout* – the input layout of the vertex shader
- *Descriptor* – a delegate that takes *MeshDx* on input and extracts the *VertexBufferBinding[]* matching the input layout of the shader.

Meshes

We can load the class *Mesh* from all file formats supported by the Assimp library [45]. The *Mesh* class contains the following data:

- *Positions : Vector3[]*
- *Normals : Vector3[]*
- *TexCoordChannels : Vector2[][]*
- *ColorChannels : Color[][]*
- *TangentBasis : (Vector3 tangent, Vector3 bitangent)[]*
- *BoneWeights : Vector4[]*
- *Indices : unit[]*
- *Dx : MeshDx*

Most of these are arrays of various types of data for each vertex. The *Mesh* will only contain the subset of these arrays that are present in the actual file. A vertex is defined by its index. For example, the vertex with index 3 has a position of *Positions[3]* and a normal of *Normals[3]*.

MeshDx is a class that mirrors this data on the GPU. It contains the same fields converted to the corresponding GPU buffers:

- *Positions : VertexBufferBinding*
- *Normals : VertexBufferBinding*
- *TextCoordChannels : VertexBufferBinding[]*
- *ColorChannels : VertexBufferBinding[]*
- *Tangents : VertexBufferBinding*
- *Binormals : VertexBufferBinding*
- *BoneWeights : VertexBufferBinding*
- *Indices : Buffer* – the index buffer

MeshDx also contains method *Update()* which updates all the GPU buffers to correspond to the underlying *Mesh*.

Textures

We can load textures from pretty much any image format. The textures can be loaded either as *Texture2D* or *ShaderResourceView*.

Lastly we can load textures as a *TextureImage*, which contains has the following properties:

- *Texture2D*
- *ShaderResourceView*
- *Width and Height*
- *NormalSize : Vector2* – the size scaled down so that *NormalSize.Length() = 1*

Custom asset loaders

To teach the game engine how to load a new type of resource, we create a new class extending the *AssetLoader<T>* base class.

- First we implement the method *Load(url : string)*. This method gets an url and returns an instance of type *T*. If there was a problem during the loading process, this method should return *null*.
- Second we implement the property *Extensions*. This property should return a list of all file extensions that are supported by this loader (a list of *string*s).

All we have to do is to define the class. Once it's defined, the game engine will find it using reflection. The only condition is that our asset loader must have a default constructor.

Custom default values

Similarly to the *AssetLoader<T>* we can implement the class *AssetDefault<T>*. For *AssetDefault<T>*, we only have to implement the method *Get(url : string)*. Even though this method gets the url, it's not supposed to load it.

This class also only needs to have a parameterless constructor so it can be found and used by the game engine.

4.8. Understanding rendering process

Rendering steps

A rendering process is defined as a list of rendering steps. A *RenderingStep* is a class containing an abstract method *Draw(context : DeviceContext)* and a virtual method *Exit(context : DeviceContext)*. Performing all of the rendering steps in succession should render the graphical object. If two graphical objects share some steps, those steps are going to be entered only once.

Inside the *Draw(...)* method, the rendering step uses the DirectX API to set the GPU state.

The *Exit(...)* method is then supposed to revert any changes to the GPU we made. However, we don't need to reset most state changes, because most state changes get later overridden by other rendering steps. In particular, we don't need to reset pixel shaders, vertex shader, textures, constant buffers, and sampler states. On the other hand, we have to reset all other changes and change any other GPU settings back to the original state.

We can see an example of a rendering step in the figure 4.1. This rendering step gets an index buffer and a list of vertex buffers and sets them in the *Draw(...)* method.

```
public class GeometryDataStep : RenderingStep
{
    VertexBufferBinding[] vertexBufferBindings;
    Buffer indexBuffer;

    public GeometryDataStep(
        Buffer indexBuffer, params
        VertexBufferBinding[] vertexBufferBindings)
    {
        this.vertexBufferBindings = vertexBufferBindings;
        this.indexBuffer = indexBuffer;
    }

    public override void Draw(DeviceContext1 context)
    {
        // Set the index buffer and vertex buffer on the GPU
        context.InputAssembler.SetIndexBuffer(
            indexBuffer, Format.R32_UInt, 0);
        context.InputAssembler.SetVertexBuffers(
            0, vertexBufferBindings);
    }
}
```

Figure 4.1: Geometry data step

Rendering graph

To render our rendering steps, we have to add them inside a rendering graph.

In the game engine, there are 4 rendering graphs. *SurfaceRendering.Graph, LightRendering.Graph, ForwardRendering.Graph,* and *EditorRendering.Graph*.

The choice of our rendering graph depends on the phase in which we want the rendering program to be performed.

The rendering is done in 4 phases:

1. Surface rendering
2. Light rendering
3. Forward rendering
4. Editor rendering

Surface rendering (1) renders on 7 output textures, each corresponding to a property of a physical surface. The textures correspond to the output hlsl structure in the figure 4.2. This is also the structure that is expected to be outputted by the surface rendering pixel shaders.

```
struct PsOut
{
    float3 position : SV_Target0;
    float3 albedo : SV_Target1;
    float3 normal : SV_Target2;
    float3 emission : SV_Target3;
    float specular : SV_Target4;
    float gloss : SV_Target5;
    float alpha : SV_Target6;
};
```

Figure 4.2: The output structure for the surface rendering

Light rendering (2) is for rendering lights. It takes the output from surface rendering and applies light equations on it. In particular, it sets the textures as indicated in the figure 4.3 for the pixel shader. Light rendering has an additive blend state which means that output colors are added together per each pixel.

```
Texture2D positionTexture : register(t0);
Texture2D albedoTexture : register(t1);
Texture2D normalTexture : register(t2);
Texture2D emissionTexture : register(t3);
Texture2D specularTexture : register(t4);
Texture2D glossTexture : register(t5);
Texture2D alphaTexture : register(t6);
```

Figure 4.3: Textures inside pixel shaders for the light rendering

Forward rendering (3) is a straight forward rendering where the pixel shaders output the pixel color directly on the back buffer.

Lastly, the editor rendering (4) is the same as forward rendering, with the only exception that it is only performed by the editor camera.

Graphics binding

To add our steps to the *RenderingGraph*, we call the method *CreateNode(...)* on the rendering graph. This method adds the rendering steps to the rendering graph and returns an instance of class *GraphicsBinding*. The *CreateNode(...)* method takes a *RenderingStep[]* as its input. Here we can see an example usage of the *CreateNode(...)* method from the *MeshRenderer* class:

```
binding = SurfaceRendering.Graph.CreateNode(
    material.Program,
    material,
    new MeshInput(Mesh.Dx),
    new DrawMethod(Draw));
```

GraphicsBinding contains two properties. Property *Graph* returns the *RenderingGraph* that was used to create it. Property *Steps* returns the current rendering steps.

To update our rendering steps, we call method *UpdateSteps(steps : RenderingStep[])* on our *GraphicsBinding*. In the following code, we can see how class *MeshRenderer* updates its binding:

```
binding.UpdateSteps(
    material.Program,
    material,
    new MeshInput(Mesh.Dx),
    new DrawMethod(Draw));
```

4.9. Creating new graphics

In this chapter we will explain how to create new graphical components from scratch.

We will assume the developer understands and knows how to use DirectX for this part.

Shaders

First we write our .hlsl shader. For simplicity, let’s use the *color.hlsl* shader from the figure 4.3. This shader has one constant buffer for color and paints all the pixels of the mesh with that color.

```
struct PsIn
{
      float4 position : SV_Position;
};

cbuffer ColorBuffer
{
      float4 color;
};

float4 Main(PsIn input) : SV_Target
{
      return color;
}
```

Figure 4.4: Pixel shader color.hlsl

Together with that shader, we will create a static class *PsColor* (figure 4.5). Whenever we are creating a class for a shader, we prefix it with “Ps”, “Vs”, “Gs”, “Hs”, or “Ds”. This class will have a static method *Set(...)*, which takes the DirectX *DeviceContext* as its parameter, and prepares the GPU for rendering using this shader. This means setting the actual shader and also things like constant buffers.

When rendering multiple objects with this shader, we expect the *Set(...)* method to be only called once.

Optionally, our class can also contain the method *SetParameters(...)* which can take arbitrary parameters that might be needed by our shader. In the example, the only parameter of our shader is the color.

If necessary, we can also include a method *Reset(...)* which resets the DirectX state changes it made. Most of the API calls are not needed to be reverted. For more information, refer to the documentation for *RenderingStep*.

```
public static class PsColor
{
    // load the .hlsl pixel shader asset
    public static PixelShader Shader =
        Assets.Get<PixelShader>("creati\\shaders\\ps\\color.hlsl");

    // create a new buffer corresponding to the color buffer
    public static Buffer ColorBuffer = new Buffer(
        Devices.Device3D, sizeof(float) * 4, ResourceUsage.Dynamic,
        BindFlags.ConstantBuffer, CpuAccessFlags.Write,
        ResourceOptionFlags.None, 0);

    // set the pixel shader using the DirectX API
    public static void Set(DeviceContext1 context)
    {
        context.PixelShader.Set(Shader);
        context.PixelShader.SetConstantBuffer(0, ColorBuffer);
    }

    // set the parameters for the pixel shader (color buffer)
    public static void SetParameters(
        DeviceContext1 context, Color4 color)
    {
        context.MapSubresource(
            ColorBuffer, MapMode.WriteDiscard,
            MapFlags.None, out var stream);
        stream.Write(color);
        context.UnmapSubresource(ColorBuffer, 0);
    }
}
```

Figure 4.5: Static class PsColor

Programs

When we bundle all of the shaders classes we need for our rendering, we create a rendering program. A rendering program is a singleton class inheriting the *RenderingStep* class. This class should always have the suffix “Program”.

This program should override the *Draw(...)* method and call *Set(...)* methods on the respective shaders. Additionally, we do any remaining GPU setup there. If any of the shaders also define the *Reset(...)* method, we need to override the *Exit(...)* method of the *RenderingStep* and call it there.

If the respective shaders have any parameters, we define a static method *SetParameters(...)* which will transitively call the *SetParameters(...)* on each shader class. Optionally, we can also define a nested class called *Instance* which will call this *SetParameters(...)* method inside its *Draw(...)* method.

We can see the resulting program in figure 4.6.

```
public sealed class SimpleProgram : RenderingStep
{
    // create the singleton instance
    private SimpleProgram() { }
    public static readonly SimpleProgram Instance = new
        SimpleProgram();

    public override void Draw(DeviceContext1 context)
    {
        // set additional rendering settings
        context.InputAssembler.PrimitiveTopology = ...;

        // set the respective shaders
        VsSimple.Set(context);
        PsColorSurface.Set(context);
    }

    public static void SetParameters(DeviceContext1 context, ...)
        => PsColorSurface.SetParameters(context, ...);
}
```

Figure 4.6: Class SimpleProgram

Constant buffers

There are a few constant buffers that are going to be used throughout most shaders. To make it easy, there is a static class *ConstantBuffers* containing them.

First constant buffer is the *Transform*, which corresponds to the hlsl cbuffer in figure 4.7. The *viewProjection* corresponds to the currently rendered camera.

```
cbuffer TransformBuffer
{
      float4x4 world;
      float4x4 viewProjection;
};
```

Figure 4.7: Transform buffer definition inside shader files

Second constant buffer is the *CameraBuffer* which corresponds to the hlsl cbuffer in figure 4.8. The *camera* is the world position of the currently rendered camera.

```
cbuffer CameraBuffer : register(b0)
{
    float3 camera;
};
```

Figure 4.8: Camera buffer definition inside shader files

The constant class *ConstantBuffers* also contains the method *UpdateTransform(transform : Matrix)*, which updates the *TransformBuffer*. The values from the cameras are updated automatically during the rendering process.

Draw method

The last step of our rendering process should be the *DrawMethod*. A draw method might do a few DirectX API calls and then call DirectX *Draw(...)* method. We can see an example of a draw method in figure 4.9.

```
public void Draw(DeviceContext1 context)
{
    if (!Visible || Mesh == null || material == null) return;

    ConstantBuffers.UpdateTransform(
        Context,
        Matrix.Translation(-origin) * Transform.World);

    context.DrawIndexed(Mesh.IndexCount, 0, 0);
}
```

Figure 4.9: Draw method

The only requirement for the draw method is that it is a method of the *Component*, not a lambda function.

We would create the corresponding *DrawMethod* like this:

```
ForwardRendering.Graph.CreateNode(...
    new DrawMethod(Draw));
```

Geometry data step

Pretty much always we need to set the geometry data on the GPU. To do that, we can use the *RenderingStep* called *GeometryDataStep*. *GeometryDataStep* takes an index buffer (class *Buffer*) and an array of *VertexBufferBinding*s. We can use it like this:

```
ForwardRendering.Graph.CreateNode(...
    new GeometryDataStep(indexBuffer, vertexBufferBindings),
    ...);
```

Mesh input

When we are rendering a mesh, we can use the *RenderingStep MeshInput*. In order for *MeshInput* to work, a *Descriptor* must be set. To get the *Descriptor*, we need to load a vertex shader as *MeshVertexShader*. To set the *Descriptor*, we call the following method:

```
context.InputAssembler.SetDescriptor(Shader.Descriptor);
```

When we have our descriptor set, we can create the *MeshInput* like this:

```
ForwardRendering.Graph.CreateNode(...
    new MeshInput(mesh.Dx),
    ...);
```

Putting it all together

Now we should have all the knowledge necessary to create a graphical component.

First, we add the rendering steps to *RenderingGraph* and get the *GraphicsBinding*. (figure 4.10)

```
protected override void Start()
{
    binding = ForwardRendering.Graph.CreateNode(
        ColorProgram,
        new MeshInput(Mesh.Dx),
        new DrawMethod(Draw));
}
```

Figure 4.10: Create a rendering binding inside the Start() method

Then we define the *Draw* method. (figure 4.11)

```
public void Draw(DeviceContext1 context)
{
    // first check if we should even render the graphical object
    if (!Visible || Mesh == null) return;

    // then update the constant buffers so our object
    // gets rendered at the correct place
    ConstantBuffers.UpdateTransform(context, Transform.World);

    // update the parameters of our ColorProgram
    ColorProgram.SetParameters(context, color);

    // and draw the mesh
    context.DrawIndexed(Mesh.IndexCount, 0, 0);
}
```

Figure 4.11: A draw method definition

Now our graphical component is going to be visible in-game. It is not, however, going to be visible in the editor yet. The reason is that we only add our rendering steps to the rendering graph in the *Start()* method, which is not getting called in the editor.

To show our graphical component in the editor, we will use the method *Component.Validate()* which is guaranteed to be called at least once. We can see how this is done in the figure 4.12.

```
public override void Validate()
{
    // Because Validate() might get multiple times,
    // we make sure to only create the binding once
    // by using the ?? operator.
    binding = binding ?? ForwardRendering.Graph.CreateNode(
        ColorProgram,
        new MeshInput(Mesh.Dx),
        new DrawMethod(Draw));
}
```

Figure 4.12: Using validate method to register graphical objects in the rendering graph

Class Rendering

If we need to find the exact information about the current window and other rendering settings, we use the static class *Rendering*. This class provides us with the following interface:

- *Width : int*
- *Height : int*
- *Aspect : float* – width / height
- *Projection : Matrix* – the *Projection* matrix of the currently rendering camera
- *View : Matrix* – the *View* matrix of the currently rendering camera
- *Viewport : Viewport* – current DirectX viewport used for rendering
- *event Resized* – triggered every time the window gets resized
- *Output : RenderingTexture* – the backbuffer that will be displayed on screen after the rendering is completed
- *ScreenDepthBuffer : RenderingTexture* – the depth buffer used together with the depth buffer
- *CurrentlyRenderedCamera : ICamera* – camera that is currently rendering the scene

RenderingTexture is a class that holds a *Texture2D* together with its various DirectX views such as *RenderTargetView, DepthStencilView,* and *ShaderResourceView*.. *RenderingTexture* contains a single method *ResizeTexture(newTexture : Texture2D)*, allowing us to automatically regenerate the views whenever the texture changes.

Custom rendering pipeline

In rare cases, we might want to create a custom rendering pipeline. To do this, we create a class that implements the *IRenderingPipeline* interface. To implement this interface, we have to write two methods:

- *Clear(viewport : Viewport)* – when rendering, first the method *Clear(...)* gets called. This method is supposed to clear the viewport provided as the parameter.
- *Render(viewport : Viewport, background : Color)* – after the viewport is cleared, the method *Render(...)* gets called. It receives the same viewport and a background color from the camera. This method will typically call *Render(...)* methods on the static classes *SurfaceRendering, LightRendering,* and *ForwardRendering*.

To better understand how the methods *Clear(...)* and *Render(...)* should look, we look at figure 4.13.

```
public void Clear(Viewport viewport)
{
    GBuffer.Clear();
    Devices.Context3D.ClearDepthStencilView(
        Rendering.ScreenDepthBuffer.DepthStencilView,
        DepthStencilClearFlags.Depth | DepthStencilClearFlags.Stencil,
        1f, 0);

    // we use the utility class ViewProgram to clear the rendering
    // output only in the area specified by the viewport
    ViewportProgram.Clear(
        Devices.Context3D, Rendering.ClearColor, viewport,
        Rendering.ScreenDepthBuffer.DepthStencilView,
        Rendering.Output.RenderTargetView);
}

public virtual void Render(Viewport viewport, Color background)
{
    // first we extract the data types we want from the
    // G-Buffer textures
    var gbufferTargets = GBuffer.RenderingTextures.Select(
        rt => rt.RenderTargetView).ToArray();
    var gbufferSources = GBuffer.RenderingTextures.Select(
        rt => rt.ShaderResourceView).ToArray();
    // we call the Render(...) static methods in the correct order
    SurfaceRendering.Render(viewport,
       Rendering.ScreenDepthBuffer.DepthStencilView, gbufferTargets);
    LightRendering.Render(viewport,
       Rendering.ScreenDepthBuffer.DepthStencilView,
       Rendering.Output.RenderTargetView);
    // we use the utility class ViewportProgram to draw a background
    // in the specified viewport
    ViewportProgram.Background(Devices.Context3D, viewport, background,
        Rendering.ScreenDepthBuffer.DepthStencilView,
        Rendering.Output.RenderTargetView);
    ForwardRendering.Render(viewport,
        Rendering.ScreenDepthBuffer.DepthStencilView,
        Rendering.Output.RenderTargetView);
}
```

Figure 4.13: A code defining the BasicPipeline

4.10. Existing shaders and programs

In this chapter we will look at the existing shader and program classes that can be used in our rendering.

VsFullscreenPlus

Perhaps the most useful one is the *VsFullscreenPlus*. This vertex captures the whole screen and doesn't need any input data. The output of this vertex shader is the structure in figure 4.14.

```
struct VsOut
{
    float4 position : SV_Position;
    float2 rel_uv : TEXCOORD; // uv relative to the viewport
    float2 abs_uv : TEXCOORD1; // uv relative to whole screen and
not just viewport
};
```

Fogire 4.14: Vertex shader output for the VsFullscreenPlus

VsDirect

This vertex shader takes only the input position (*Vector3*) and passes them directly to the next shader stage without changing it.

VsSimple

This vertex shader takes position and normal and transforms it using the *ConstantBuffers.TransformBuffer* to the next stage. For exact input and output information, see the *creati/shaders/vs/simple.hlsl*.

VsSimple

This vertex shader takes position, normal, uv, tangent, and binormal, and transforms them using the *ConstantBuffers.TransformBuffer* to the next stage. For exact input and output information, see the *creati/shaders/vs/complex.hlsl*.

PsColor and PsColorSurface

These two pixel shaders simply color all the pixels to the color specified by its respective *SetParameters(...)* methods. The difference between the two is that the *PsColor* is for forward rendering while the *PsColorSurface* is for surface rendering and requires additional information from the previous stage (world position and normal).

PsTexture

This pixel shader requires uv coordinates from the previous stage and then renders the texture set by its *SetParameters(...)* method.

GizmoProgram and QuadProgram

Lastly there is a *GizmoProgram* and *QuadProgram*. The tutorial for them is in chapter *4.10. Editor gizmos*.

4.11. Working with the inspector

There are two methods that the editor might call on the component (or any *IInspectable class*).

First is the method *Reset()*. *Reset()* is called when a component is first added through the inspector and whenever the user right-clicks the component and selects *Reset*. The implementation of *Reset()* should put the object in some default state.

Second is the component *Validate()*. *Validate()* is called whenever any value of the component is changed through the editor. *Validate()* should ensure that the component is in a stable state. *Validate()* is also called whenever the object is first created or when the object gets deserialized. That means that *Validate()* is guaranteed to get called at least once while in the editor.

Components are sorted under various menu items in the inspector. To control which menu item our component goes under, we can use the attribute *[GroupName(name : string)]*

4.12. Editor gizmos

When we want to render some additional information only in the editor, we add the rendering steps to the *EditorRendering.Graph*. Because this is only rendered while in the editor, it would be pointless to add it inside the *Start()* method.

Gizmos

To create gizmos, we use the class *GizmoMeshBuilder*. This class builds a mesh with only *Positions* and *Colors*.

The *GizmoMeshBuilder* has following methods and properties:

- *Append(position : Vector3)* – adds a new vertex with at the specified position
- *NextCurve()* – ends the current curve and starts building a new curve (cuts the mesh)
- *BrushColor : Color* – sets the color that the new vertices will have
- *Finish()* – constructs the *Mesh* and returns it

An example of a *Mesh* being built inside a *GizmoMeshBuilder* is in figure 4.15.

```
GizmoLinesBuilder builder = new GizmoLinesBuilder();

builder.BrushColor = Color.Red;
builder.Append(Vector3.Zero);
builder.Append(Vector3.UnitX);

builder.NextCurve();

builder.BrushColor = Color.Green;
builder.Append(Vector3.Zero);
builder.Append(Vector3.UnitY);

builder.NextCurve();

builder.BrushColor = Color.Blue;
builder.Append(Vector3.Zero);
builder.Append(Vector3.UnitZ);

Axis = builder.Finish();
```

Figure 4.15: Generating a gizmo mesh using the GizmoMeshBuilder

To render our gizmo *Mesh*, we use the following program in figure 4.16.

```
public override void Validate()
{
    gizmo = gizmo ?? EditorRendering.LateGraph.CreateNode(
        GizmoProgram.Lines,
        Settings.Override, // an optional rendering step
        new MeshInput(Axis.Dx),
        new DrawMethod(Gizmo));
}

public void Gizmo(DeviceContext1 context)
{
    // check if the this Component's GameObject is selected
    // in the editor
    if (!EditorRendering.Selected(this)) return;

    // update the position of where our gizmo will be rendered
    ConstantBuffers.UpdateTransform(context, Transform.World);
    // draw the gizmo
    context.DrawIndexed(Axis.IndexCount, 0, 0);
}
```

Figure 4.16: Rendering the gizmo mesh

First we notice the line *Gizmo.Lines* – this is the actual rendering program. There are three rendering programs for gizmos: *Gizmo.Lines, Gizmo.Meshes,* and *Gizmo.Points*.

Next we will notice the rendering step *Settings.Override*. This step is optional and causes our gizmo to be drawn on top of everything else.

Lastly we notice the *EditorRendering.Selected(this)*. This method returns true only when the game object is selected by the editor. Alternatively there also exists a method *EditorRendering.Hovered(this)*, which only returns true when the component is being hovered over by the mouse inside the inspector. We can use these two methods to draw our gizmo conditionally.

As a result we get the image in figure 4.17. The gizmo is the green, red, and blue line showing it's position in space.

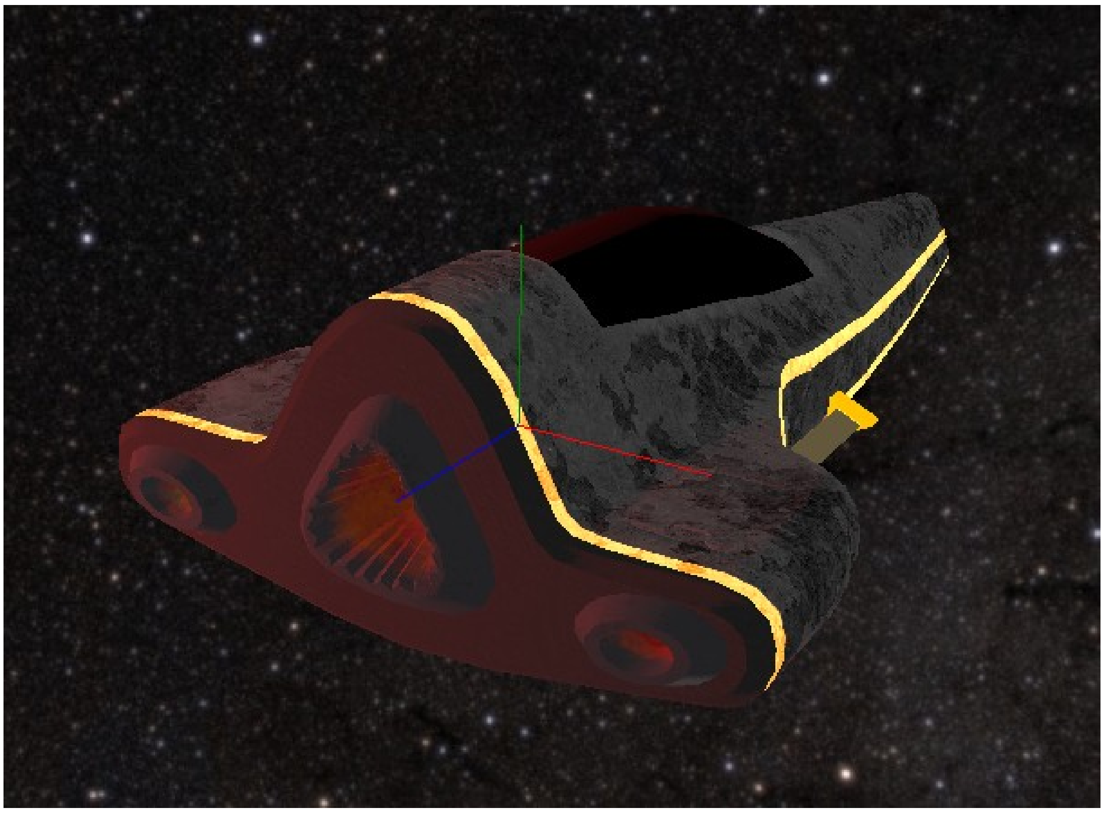


Figure 4.17: Gizmo that shows a center of our game object

Floating images

We can also create gizmos from 2D images that will turn towards the camera. Creating these kinds of gizmos is easier. First, we define the rendering steps as in the figure 4.18.

```
static TextureImage gizmoImage = Assets.Get<TextureImage>("...");

public override void Validate()
{
    gizmo = gizmo ?? EditorRendering.Graph.CreateNode(
            QuadProgram.Instance,
            new DrawMethod(Gizmo));
}

public void Gizmo(DeviceContext1 context)
{
    if (gizmoImage == null)
        return;

    // simply set the parameters
    QuadProgram.SetParameters(context,
        gizmoImage.ShaderResourceView, // the image
        Transform.World.TranslationVector, // the position
        gizmoImage.NormalSize); // the size
    // and draw
    context.Draw(4, 0);
}
```

Figure 4.18: Using the QuadProgram

Using this rendering program is pretty straight forward. The most difficult part is the *QuadProgram.SetParameters*. As a result we will get something like in the image 4.19.

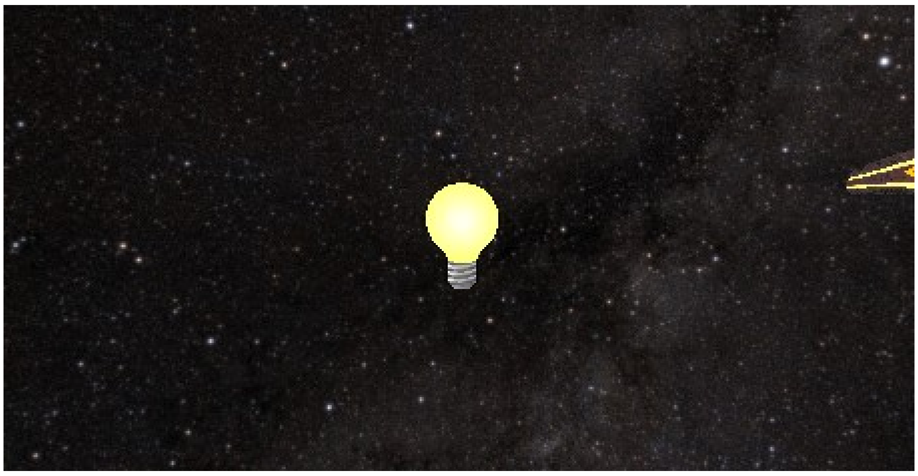


Figure 4.19: The result of the QuadProgram

4.13. Creating custom systems

Let's say we want to create a system that manages all *Health* components and for example shows them sorted on the screen.

First, we create a static class that will act as the manager. Let’s call it *HealthSystem*.

```
public static class HealthSystem {...}
```

Scene static properties

Let’s say we want some information to be static per scene. We will use the class *SceneStatic<T>* to achieve that. Class *SceneStatic<T>* holds a value of type *T* per each scene. We can then use the property *Current* to retrieve the value for the current scene. The constructor for *SceneStatic<T>* takes a *Function<T>* which is used to initialize the internal variable for each scene.

As an example, let’s create an integer which holds the sum of all health points left. To create that, we would write something like in figure 4.20. First, we create our *SceneStatic<T>* property. This property should be defined as static and private. We will set the initial value to 0. Together with that, we create a public static accessor that uses the property *Current* of the class *SceneStatic* to retrieve the value corresponding to the current scene. By using the accessor pattern, we have created a property that appears to be static but is actually scene static.

```
static SceneStatic<int> totalHealth = new SceneStatic<int>(
    () => 0);
public static int TotalHealth => totalHealth.Current;
```

Figure 4.20: Defining a scene static variable

Component collections

First, we need to store our components in some collection. We will use the class *UnorderedComponentList<T>* to hold the components. *It is a* collection that extends the *List<T>*. The type argument *T* must always be a subclass of the *Component* class. This collection adds 2 useful methods:

1. *RemoveFast(value : T)* – this method finds the value in the collection and then removes it. To remove it, it swaps it with the last element and reduces size by one. This destroys the ordering, hence “unordered”.
2. *FilterAlive()* – this enumerates over all components which are alive. During the enumeration it also removes all the dead components from the list. This allows us to simply iterate over *FilterAlive()* every frame and not do any additional management.

Alternatively we could also use regular *List<T>* and use the extension methods that provide the two same functions for all collections. The only difference is that the *FilterAlive()* extension method needs a *Function<bool>* as its parameter to determine whether the element is dead or not.

```
using HL = UnorderedComponentList<HealthComponent>;
...
static SceneStatic<HL> components = new SceneStatic<HL>(
    () => new HL());
public static int Components=> components.Current;
```

Figure 4.21: Scene static collection of HealthComponents

We will define our component collection using the *UnorderedComponentList<T>*. We will make this collection scene static because we want each scene to only have one. The resulting code might look like the code in figure 4.21.

Scene static events

Let's say we want to create an event that gets triggered when the total health goes down to 0. To do that we will use the class *CreatiEvent*. *CreatiEvent* comes in 2 variants: non-generic and generic. The non-generic variant calls event handlers without any parameters. The generic variant (*CreatiEvent<T>*) calls event handlers with one parameter of the type *T*.

The *CreatiEvent* has the following interface:

- *AddHandler(handler : delegate)*
- *RemoveHandler(handler : delegate)*
- *Invoke(invocation : Action<T>)* – the *invocation* is a method that takes our handler delegate and invokes it
- *Invoke()* – we can only use this method in the non-generic variant

The *CreatiEvent* is smart. Whenever we try to invoke it, it checks whether the components holding the event handler delegates are alive, and if they are not, it removes them. This only works if the event handler delegate is a method directly inside a component.

When we create the event, we will use a similar accessor pattern as we have used for the previous two scene static variables, except for events.

```
private static SceneStatic<CreatiEvent> allDead
    = new SceneStatic<CreatiEvent>(() => new CreatiEvent());

public static event Action AllDead
{
    add => allDead.Current.AddHandler(value);
    remove => allDead.Current.RemoveHandler(value);
}
```

Accessing scene static properties and events

Accessing scene static properties and events is really simple. For example, let's say we want our health components to add themselves to the *HealthComponents* collection. We can simply use the code in figure 4.22 to do that. Additionally, this code also shows how to react to our custom event.

```
protected override void Start()
{
    HealthSystem.Components.Add(this);
    HealthSystem.AllDead += AllDead;
}

void AllDead()
{
    ...
}
```

Figure 4.22: Adding the HealthComponent to the HealthSystem and registering an event handler for the CreatiEvent

Modifying our game loop

To integrate our *HealthSystem* directly to the game loop, we will call the method *Loop.Modify(...)*. This method takes as a parameter a delegate which receives the list of actions of the loop and performs the modifications on it.

To know how exactly we want to modify our loop, we need to know how the original game loop knows. The original game loop is shown in the figure 4.23.

```
private static readonly List<Action> loop = new List<Action>()
{
    SceneManager.ManageGameObjects,
    Time.Update,
    Input.Update,
    Events.CallEarlyUpdate,
    Events.CallUpdate,
    Physics.Update,
    Events.CallLateUpdate,
    Rendering.RenderAllCameras,
    GameSurface.Present,
    Wpf.ProcessMessages,
};
```

Figure 4.23: Default game loop

Now that we know how our game loop works, we decide that we want to update our *HealthSystem* after the *Events.CallUpdate*. First we will write our static *Update* method inside of the *HealthSystem* and then we will add it to the place we chose. The code for this is shown in the figure 4.24. To perform the modification exactly once, we call it inside the static constructor

```
static HealthSystem
{
    Loop.Modify(loop => loop.Insert(
        loop.IndexOf(Events.CallUpdate), // where to insert
        Update)); // the method to insert
}

public static void Update()
{
    // update the sum of healths
    // if it's zero call the event
}
```

Figure 4.24: Modifying the game loop

4.14. Input

For input we use the static class *Input*. The class *Input* abstracts away the physical key presses under *string* names. It provides us with the following interface:

- *SetBinding(name : string, key : VirtualKey)* – This method binds a *VirtualKey* to a *string* name. The *VirtualKey* is an enumeration of all possible keyboard and mouse keys and buttons (including scroll wheel).
- *RemoveBinding(name : string)* – removes the key binding
- *IsKeyDown(name : string)* – returns true if the key is currently down
- *IsKeyUp(name : string)* – returns true if the key is currently up
- *IsKeyPressed(name : string)* – returns true if the key got just pressed
- *MouseInBounds()* – returns true if the mouse is inside the window
- *MousePixelPosition()* – returns the position of mouse in pixels, relative to bottom left corner
- *NormalMousePosition()* – returns the position of the mouse relative to the center of the window. Center of the window being (0,0), top right corner being (1,1), and bottom left corner being (-1, -1).
- *NextKeyPress* – is an event that gets triggered the next time any key is pressed. After the event gets triggered, it unbinds all the event handlers.

The key bindings are typically going to be set using the *InputMap* component.

4.15. User interface

To create a custom interface, we create a new user control in visual studio. We then change the underlying class of both files (.xaml and .xaml.cs) to the *Creati.GridComponent*. In the xaml file, we also have to add a reference to the *Creati* namespace. Figure 4.25 shows the partial declarations of those classes.

```
// OurUI.xaml.cs
public partial class OurUI : GridComponent {...}

// OurUI.xaml
<creati:GridComponent x:Class="game.OurUI"
             xmlns:creati="clr-namespace:Creati;assembly=Creati"
             ...>
```

Figure 4.25: Creating a custom GridComponent

Now we just create the user interface we want using the visual studio xaml editor. Because the *GridComponent* inherits the *Component*, we can override methods such as *Start()*, *Validate()*, etc. We can work with it like with a regular component.

4.16. Physics

Class *PhysicalSimulation* contains the current physical simulation using the *Bullet Physics* library. We can access the *PhysicalSimulation* for the current scene using the scene static property *Physics.Simulation*.

The *PhysicsSimulation* provides the following interface:

- *Step : TimeSpan* – the step of the simulation
- *Gravity : Vector3* – the gravitational force of the simulation; default is a downward force of 9.807f
- *BeforeSimulation : CreatiEvent* – gets called every frame before the simulation
- *AfterSimulation : CreatiEvent* – gets called every frame after the simulation
- *BeforeStep : CreatiEvent* – gets called before every simulation step
- *World : DiscreteDynamicsWorld* – the underlying *Bullet Physics* simulation
- *AddRigidBody(body : RigidBody)* – adds a *BulletPhysics.RigidBody* to the simulation
- *RemoveRigidBody(body : RigidBody)* – removes a rigid body from the simulation

4.17. Serialization

By default, the serializer serializes only public fields. If we wish to serialize private fields, we can use the attribute *[SerializeField]*.

The interface *IDeserializable* allows us to implement a custom behavior after deserializing. The interface contains a single method called *Deserialized()*. If any class implementing the interface gets deserialized, the method *Deserialized()* gets called.

4.18. Editors

The game engine provides a default editor for primitive types such as *strings, bools, enums,* and numerical types. It provides a default drag and drop editor for the class *GameObject* and for all the classes that can be loaded from assets. It also contains special editors for structures *Vector3*, *RectangleF*, *Color*, and for the enum *VirtualKey*. For collection there is the *ObservableCollectionEditor*. All structures that don't have their special editor will have their editor generated recursively.

By default, only public fields have an editor inside the inspector. We can change this by adding *[ShowEditor]* or *[HideEditor]* attributes to our class members. We can use these attributes on fields, members, and even parameterless methods.

If we use the *[ShowEditor]* on a parameterless method, the inspector generates a button that calls that method on press. This could be used for example for various randomized components (e.g. terrain generator).

Using properties as views

```
// Hide the actual data
[SerializeField, HideEditor]
private PointLightProgram.Data data = new PointLightProgram.Data();

// Provide an interface to access the data through property editors
[ShowEditor]
public Color Color
{
    get => new Color(data.Color);
    set => data.Color = value.ToColor3();
}
[ShowEditor]
public float ConstantFactor
{
    get => data.C1;
    set => data.C1 = value;
}
[ShowEditor]
public float LinearFactor
{
    get => data.C2;
    set => data.C2 = value;
}
[ShowEditor]
public float QuadraticFactor
{
    get => data.C3;
    set => data.C3 = value;
}
```

Figure 4.26: Using properties as views

We can use properties to change the way we look at our data through editors. For example in the figure 4.26, we have a structure called *data*, but we create direct assessors *Color, ConstantFactor, LinearFactor,* and *QuadraticFactor*. These direct assessors also rename the fields which would otherwise be inappropriate. Furthermore, the color accessor changes the type of color from *Color3* to *Color*, which is the one we have a default editor for.

Another example of this is inside the class *Camera*, where the internal property for field of view is in radians but the *FieldOfView* property used as a view is in degrees.

Overriding editors

If we are not happy with the default editor for the type, we can override it by instantiating the editor inside the attribute for the class member whose editor we want to change. Doing this will also force the editor to be generated for that member.

An example of this is in the figure 4.27. The only editors provided by the game engine which are not set as the default editor are the slider editors. There is a slider editor for each numerical type.

```
[IntSliderEditor(Min = 0, Max = 500)]
public int Health;
```

Figure 4.27: Overriding the default editor

Creating custom editors

To create a custom editor, we extend the class *ValueEditor* and implement the method *EditValue(context : ValueContext)*. This method takes a *ValueContext* as a parameter and returns a *FrameworkElement* of the editor.

ValueContext is a class that allows us to get and set the value of arbitrary fields and properties of classes and structs. The *ValueContext* provides the following interface:

- *Type : Type* – the type of the field or property
- *DisplayName : string* – the name of the field or property
- *event UpdateVisual* – an event that gets called when the underlying value has changed and our editor should update the visual elements to reflect that. A parameter of this event is the new value in the form of an *object*
- *ValueChanged(newValue : object)* – a method that we call whenever the user changed the value inside the editor and we want this value to get transferred to the underlying object

Optionally, we can also mark our editor with the *[DefaultFor(type)]* attribute so that it is used by default for all fields and properties of that type.

We can see how such an editor might look in figure 4.29.

```
[DefaultFor(typeof(bool))]
public class BoolEditor : ValueEditor
{
    public override FrameworkElement EditValue(ValueContext context)
    {
        var checkBox = new CheckBox();
        checkBox.VerticalAlignment = VerticalAlignment.Center;
        checkBox.LayoutTransform = new ScaleTransform(1.2, 1.2);

        checkBox.Checked += (s, e) => context.ValueChanged(true);
        checkBox.Unchecked += (s, e) => context.ValueChanged(false);
        context.UpdateVisual += (v) => checkBox.IsChecked = (bool)v;

        return EditorHelper.NameWrap(checkBox, context);
    }
}
```

Figure 4.29: BoolEditor implementation

4.19. Components

In this chapter we will discuss the components included in the game engine.

Transform

Transform is the only component default to all game objects. We can access the transform component using the property *Transform* on either *GameObject* or *Component*. It is single handedly the most important component.

Transform has the following properties:

- *Position : Vector3*
- *Scale : Vector3*
- *Rotation : Quaternion*
- *RotationVector : YawPitchRoll* – this property can only be set in the editor or when first creating the game object. That is because the *Quaternion Rotation* gets initialized inside the *Start()* using this value.
- *Forward, Back, Up, Down, Left, Right : Vector3* – properties that return directions relative to the current rotation.

The *YawPitchRoll* of the rotation vector is a simple structure with float properties *Ya, Pi*, and *Ro*. This structure is structurally equivalent to *Vector3*. The only purpose for its existence is to rename the element names inside the editor.

Camera

First is the *Camera* component. *Camera* has properties the following properties:

- *FieldOfView : float*
- *Background : Color*
- *Near,* and *Far : float*
- *ViewportRectangle* – controls where the camera gets rendered on the screen in normalized units, i.e. from (0,0) to (1,1)
- *Depth* – controls the order in which cameras get rendered; lowest *Depth* camera gets rendered on top

- *RenderingPipeline : IRenderingPipeline* – the rendering pipeline that the camera will use for rendering

Point light

PointLight creates a point light source on the scene. The color and intensity is controlled by the property *Color*. The falloff factor is controlled by properties *ConstantFactor, LinearFactor,* and *QuadraticFactor*.

Ambient light

Component *AmbientLight* creates a source of ambient light on the scene. Each scene should have exactly one *AmbientLight* component. *AmbientLight* has a single property *Color* which controls its color and intensity.

Skybox

Component *Skybox* renders a skybox around the sphere. The skybox contains a single property *Image*, which corresponds to the skybox. The skybox gets rendered on a sphere around the player and the skybox image should look like in the picture 4.30.

Figure 4.30: Image used for skybox

Image renderer

Component *ImageRenderer* renders a 2D image in 3D space, always facing the player. It has properties *Image* and *Size*.

Mesh renderer

Component *MeshRenderer* can render meshes. It has following properties:

- *Mesh*
- *Material : Material*
- *Origin : Vector3* – controls the position of the mesh relative to the center point of the object

Material controls the shaders and other DirectX settings in the rendering process of the mesh.
Class *Material* inherits the *RenderingStep* class and contains a single property called *Program : RenderingStep*. The *Program* rendering step should set the DirectX settings common for all instances of this material, such as shaders. The *Material*

RenderingStep itself should set all the material instance settings, i.e. things that differ between different instances of the material, such as textures and constant buffers. The material will be rendered inside the *SurfaceRenderer*. We can see an example of a simple material in figure 4.31.

```
public sealed class SimpleMaterial : Material
{
    public Color color = Color.White;
    public Color emission = Color.Black;

    [FloatSliderEditor(DecimalPlaces = 2, Min = 0, Max = 1)]
    public float gloss = 0f;

    [FloatSliderEditor(DecimalPlaces = 2, Min = 0.01, Max = 512)]
    public float specularPower = 64f;

    public override RenderingStep Program =>
        SimpleProgram.Instance;

    public override void Draw(DeviceContext1 context)
    {
        SimpleProgram.SetParameters(context,
            color.ToColor3(),
            emission.ToColor3(),
            Gloss,
            specularPower);
    }
}
```

Figure 4.31: SimpleMaterial implementation

This program uses the *SimpleProgram* as the base program while the *SimpleMaterial* itself sets the data unique to each instance of the material, such as *color, emission, gloss,* and *specularPower*.

Particle generator

Component *ParticleGenerator* can generate particles. It has the following settings:

- *ParticleSize : float*
- *ParticleLifeTime : float* – particle lifetime in seconds
- *Velocity : Vector3* – controls the base velocity of particles
- *VelocityVariance : Vector3* – adds a random vector within a box defined by corners -*VelocityVariance* and *VelocityVariance* to the *Velocity*
- *Spread : Vector3* – the particles will get spawned in a box defined by corners -*Spread* and *Spread* around the game object's position
- *EmitPerSecond : float* – how many particles will the particle generator emit per second
- *Material : ParticleMaterial*

To spawn our particles, we have two options. First, we can control the *EmitPerSecond* property and second we can call the method *EmitRandomParticles(amount : int)*.

The *ParticleMaterial* of the *ParticleGenerator* is simpler than the material for meshes. It only contains properties *Texture1, Texture2* and *PixelShader*. When the particle generator is being rendered, the textures and pixel shader used for the rendering. If no pixel shader is provided, a default pixel shader which simply draws the first texture fading away is used.

The pixel shader must output a single color and take as an input the structure in figure in figure 4.32. The *fade* is a value between 0 and 1, the *fade* starts as 0 at the particle's lifetime and slowly changes to 1 as the particle dies.

```
struct PsIn
{
	float4 position : SV_Position;
	float4 worldPos : WORLDPOS;
	float2 uv : TEXCOORD;
	float fade : FOG;
};
```
Figure 4.32: Input structure for the pixel shader used in the ParticleMaterial

Rigid body

Component *RigidBody* creates a *Bullet Physics* rigid body (further referenced to as BPRB) and attaches it to the physical simulation. The BPRB can be directly accessed through the property *BulletRigidBody*.

The component *RigidBody* also contains following properties:

- *Mass : float*
- *Friction : float*
- *RollingFriction : float*
- *Bounciness : float*
- *GhostObject : bool* – when *GhostObject* is true, the rigid body still registers collisions but doesn't get affected by them. This can be used for example to define areas that trigger some action when the player enters.
- *InitialVelocity : Vector3* – a simple way to set an initial velocity for the rigid body
- *DragConst, DragLinear, DragQuadratic : float* – these properties control the drag generated by the object, making it slow down over time
- *AngularDragConst, AngularDragLinear, AngularDragQuadratic* – these properties control the rotational drag generated by the object spinning, making the object stop spinning over time

The *RigidBody* component also contains two methods for manipulating it. *ApplyForce(force : Vector3)* and *ApplyTorque(torque : Vector3)*. These methods do exactly that – apply force and torque to the rigid body.

Collider components

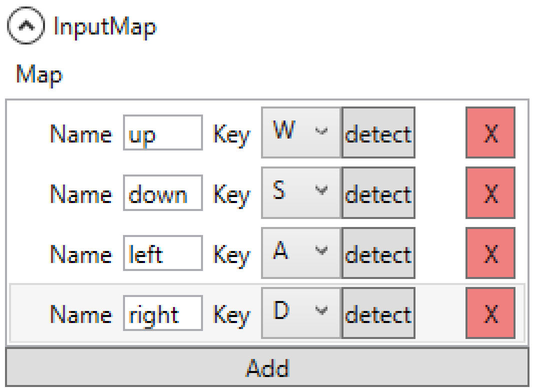
When the *Start()* method is called on the *RigidBody* component, it tries to find any components on the same game object inheriting the abstract class *ColliderComponent*. *ColliderComponent* must implement a single method *GetShape()*, which returns a shape from *Bullet Physics*. The *RigidBody* component then uses that shape inside the physical simulation for collisions.

The *ColliderComponent* also provides us with an event *Collision*, which is triggered every time the rigid body collides with another object. This event provides information about the other *ColliderComponent* as well as the world position of the collision.

There are 2 colliders provided by the game engine: *BoxCollider* and *SphereCollider*. Both of those colliders contain a property *Origin* that controls the center point of the collider. On top of that, the *SphereCollider* contains a *Radius : float* property while the *BoxCollider* contains a *Box : Vector3* property. Hovering over these collider components in the inspector will show their respective collision shapes.

Input map

The component *InputMap* allows us to easily map real keys (*enum VirtualKey*) to the string names used by the *InputClass*. It doesn't have any public properties but provides a great editor interface for setting up the key bindings as we can see in figure 4.33.



InputMap

Map

Name	up	Key	W
Name	down	Key	S
Name	left	Key	A
Name	right	Key	D

Figure 4.33: InputMap in the editor

Look at

A simple utility component *LookAt* has a single property *Target : GameObject*. This component will simply keep turning towards the *Target*'s position.

Follow on string

The utility component *FollowOnString* has two properties. *Target : GameObject* and *MaximumDistance : float*. Whenever the distance between the current game object and the target game object is larger than the maximum distance, it will move towards the target game object's position to make it equal to that maximum distance.

4.20. Debug

To debug our scripts, we open the game project in Visual Studio, select the *Debug → Attach to Process…* (Ctrl + Alt + P), and then find the editor in the list of projects. After we have done that, we can use breakpoints and all other debugging tools inside the visual studio.

5. Demo tutorial

In this chapter, we will look at how to create the demo provided together with the game engine.

5.1. Demo walkthrough

Inside the demo we fly a ship and try to shoot asteroids while avoiding getting hit by them. We can see a screenshot of the demo in the picture 5.1. We have a total of 3 health.

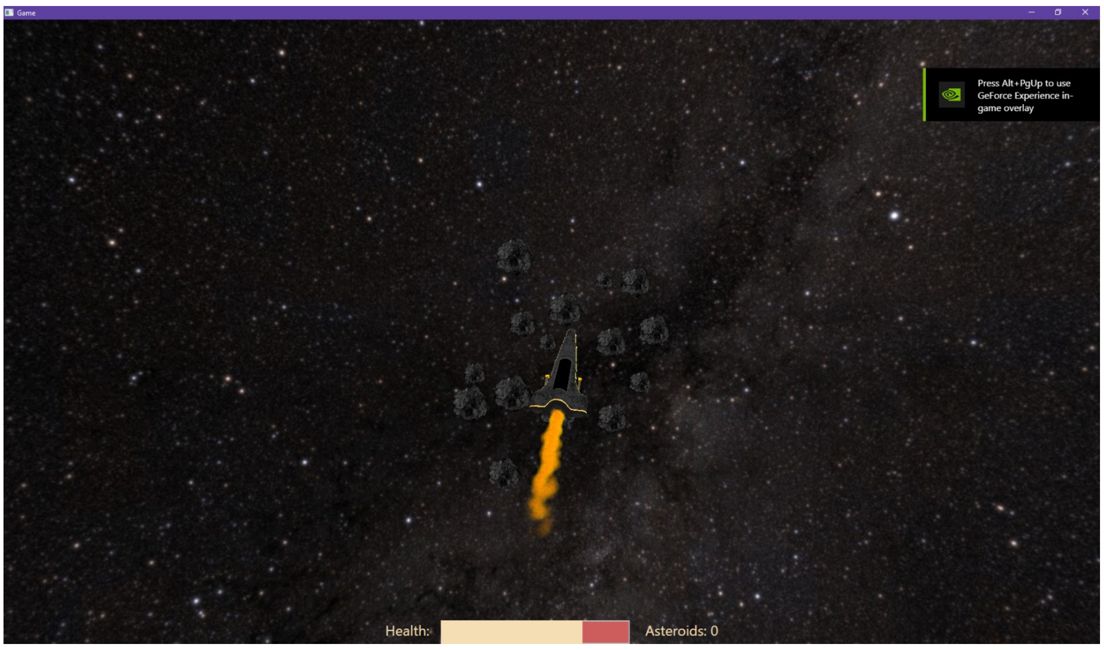


Figure 5.1: A screenshot from the demo

We control the ship with the following keys:

- *W, S, A, D, Q, E* - rotate ship in various directions
- *Space, Ctrl* - thrust forward or back
- *X* - shoot laser

When we lose all of our health, the demo will take us to the game over screen, showing the score and giving a button to restart the game (figure 5.2).

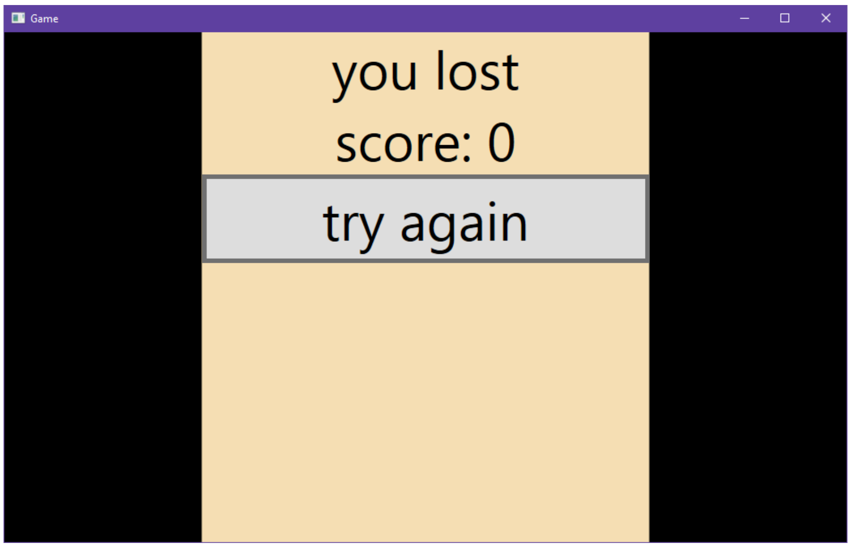


Figure 5.2: Game over screen

Now we will talk about how to create such a game. Code snippets will be shown throughout the tutorial. It would be however impractical to show all of the code. For better experience, the reader can follow the demo game code in the included demo.

5.2. Ship

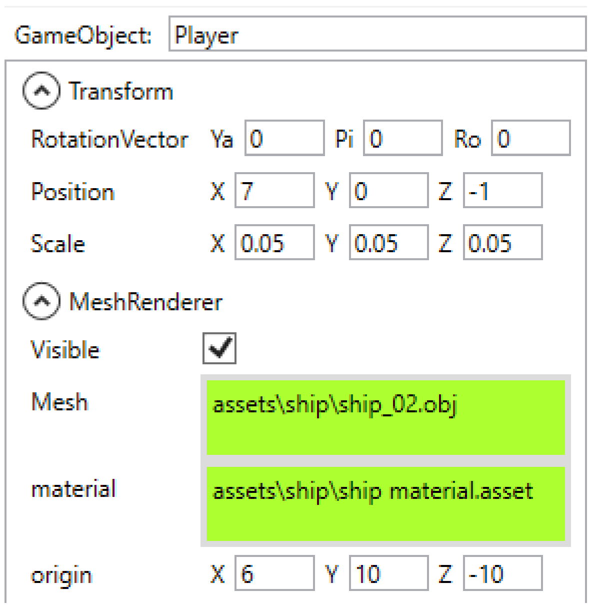
First we create a game object named "Player". We add a mesh renderer and set the mesh to our ship by dragging the mesh file from the asset browser to the *Mesh* property. Since our mesh is way too large, we scale it down by changing the *Scale* to 0.05 in all dimensions.

Next we set the ship's center of mass to the proper place. To do that we play around with the *origin* property of the *MeshRenderer* until the ship is positioned around the center of mass in a way we want.

Then we create a material for our ship. We right click in the asset browser and select *ComplexMaterial*. By double clicking the asset, we can rename it and set up the textures prepared for the ship.

Lastly we set the material for the ship's *MeshRenderer* using the drag and drop functionality.

Figure 5.3 shows how our ship currently looks in the inspector and figure 5.4 shows how the ship looks in the editor's world renderer.



GameObject: Player

Transform

RotationVector	Ya	0	Pi	0	Ro	0
Position	X	7	Y	0	Z	-1
Scale	X	0.05	Y	0.05	Z	0.05

MeshRenderer

Visible	✓
Mesh	assets\ship\ship_02.obj
material	assets\ship\ship material.asset
origin	X 6 Y 10 Z -10

Figure 5.3: Player game object in the inspector

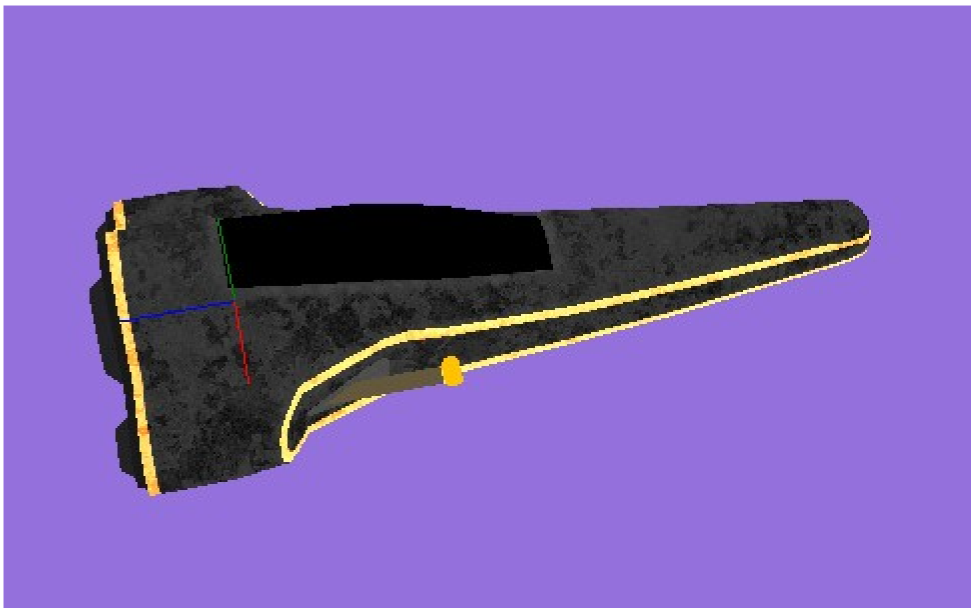


Figure 5.4: Player game object in the world renderer

Next we create 3 children objects under the ship. First game object is for the thruster and the other 2 are for the cannons. By manipulating the *Transform* component we place them in the desired location. We want the children's objects to be at the place where the thruster will generate particles and where the cannons will shoot from. Figure 5.5 shows the resulting object hierarchy.

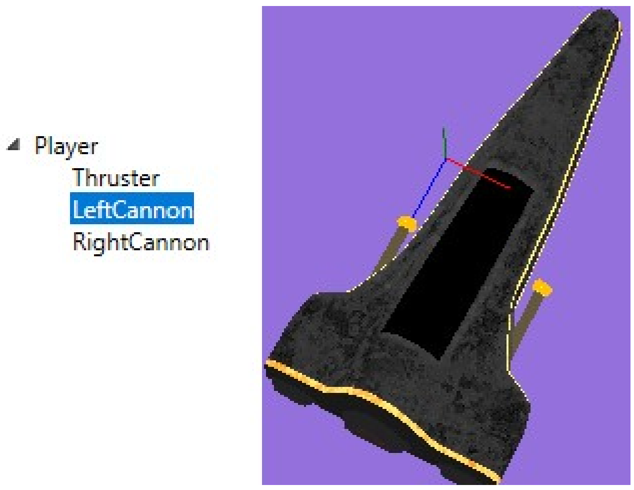
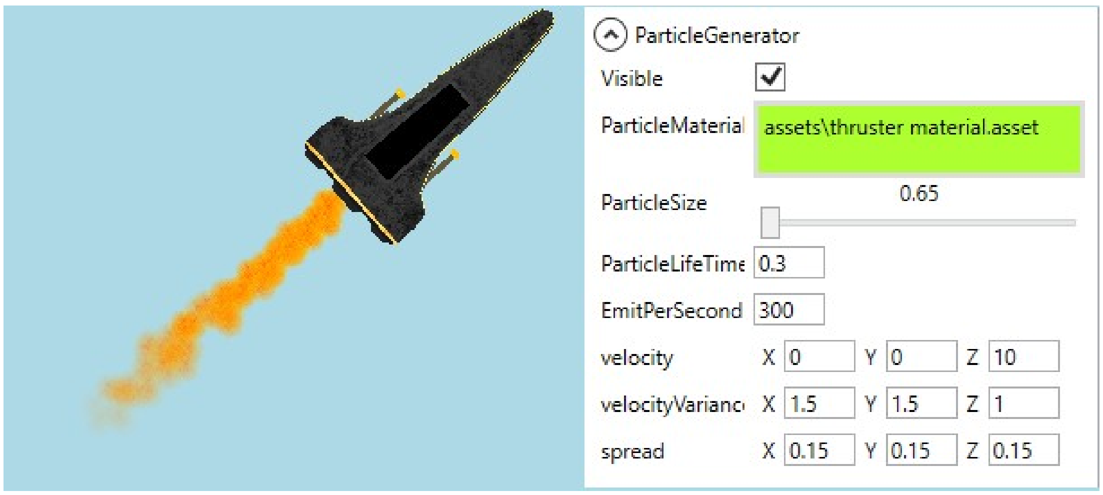


Figure 5.5: Game object hierarchy

Next we add a particle generator under the *Thruster* game object. First we create a new *ParticleMaterial* asset and set the first texture to an image of a fire particle. Next we click *Play* to debug our game and play around with the *ParticleGenerator* until the thruster looks good. The debugging process is depicted at the picture in figure 5.6.



ParticleGenerator

Visible	✓					
ParticleMaterial	assets\thruster material.asset					
ParticleSize	0.65					
ParticleLifeTime	0.3					
EmitPerSecond	300					
velocity	X	0	Y	0	Z	10
velocityVariance	X	1.5	Y	1.5	Z	1
spread	X	0.15	Y	0.15	Z	0.15

Figure 5.6: Thruster debugging

5.3. Skybox

Next step is to add a skybox to the game. We create a game object named "Skybox" and put a single component of type *Skybox* inside. We will use the skybox texture provided by the game engine with the path *creati/textures/starmap_4k.png*. With the skybox, our world render will look like this (figure 5.7).

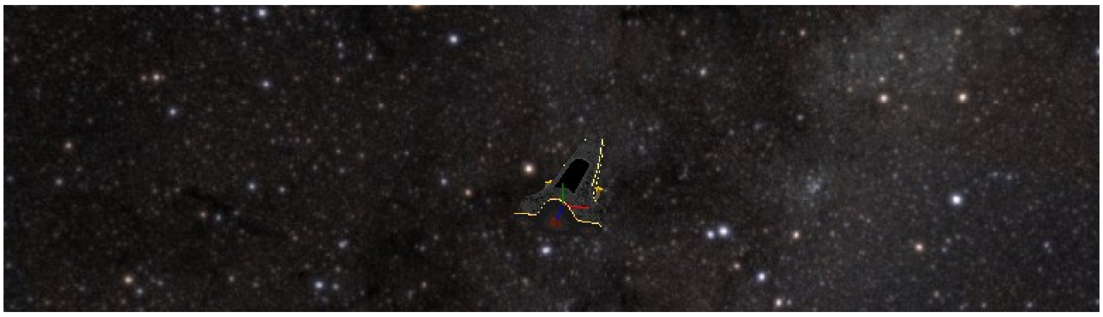
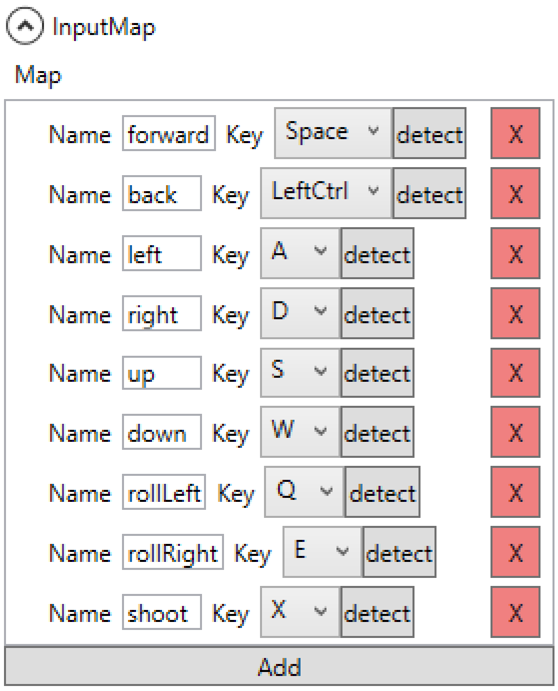


Figure 5.7: Skybox

5.4. Controls

Next we need to make our ship controllable. First we use the component *InputMap* to set up the key bindings. It doesn't matter where we place this component so we might as well put it under the "Player" game object. We will set up our keybindings like shown in the picture 5.8.


InputMap

Map

Name	forward	Key	Space	detect	X
Name	back	Key	LeftCtrl	detect	X
Name	left	Key	A	detect	X
Name	right	Key	D	detect	X
Name	up	Key	S	detect	X
Name	down	Key	W	detect	X
Name	rollLeft	Key	Q	detect	X
Name	rollRight	Key	E	detect	X
Name	shoot	Key	X	detect	X

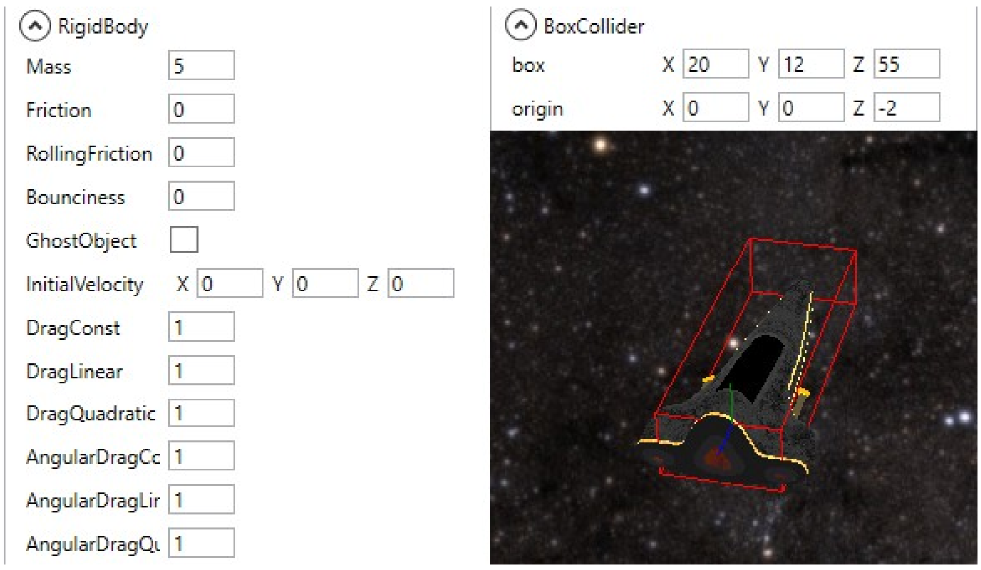
Add

Figure 5.8: Input map with the keybindings for our demo

5.5. Physics

In order for our ship to fly, we need to make it a physical object. To do that, we first add a component *RigidBody* to the "Player" game object. We set the mass to 5 because we want the ship to be a little bit heavier than the asteroids we will later generate with mass 1. We also set all the **Drag* properties to 1 so our ship doesn't fly off into infinity. The ship slowing down will also make it more intuitive to control.

We also add a *BoxCollider* to the ship and play around with its size until it fits our mesh well enough. We can hover over the *BoxCollider* component in the inspector to see the collision shape. The settings of the rigid body and the collider are shown in the picture 5.9.



RigidBody

Mass	5
Friction	0
RollingFriction	0
Bounciness	0
GhostObject	☐
InitialVelocity	X 0 Y 0 Z 0
DragConst	1
DragLinear	1
DragQuadratic	1
AngularDragCc	1
AngularDragLir	1
AngularDragQι	1

BoxCollider

	X	Y	Z
box	20	12	55
origin	0	0	-2

Figure 5.9: Rigid body and box collider around our ship

5.6. Camera

Now we will need to make our camera follow our ship. Since the asteroids will always come only from one side, we will make the camera always point to that direction.

For the camera controller we will need to create a new script. In the menu, we click *Project → Open solution in visual studio*. We then create a new component class *ShipCamera* and set the group name to "Custom" (class attribute *[GroupName("Custom")]*.

We want the camera to follow our "Player" game object so we add a field *TargetObject : GameObject*. We don't want this field to be accessible from outside so we make it private and add the attribute *[SerializeField]* so it gets serialized and shown in the editor.

The idea is that the camera will always face in the same direction (forward) and smoothly follow the position defined by fields *Height* and *Distance*. Lastly we will add a field *smoothSpeed : float*, which will control how much of the camera smoothing is done.

We want to update the camera position every frame after the physical simulation gets updated. For that there we will use the event called *LateUpdate*. The code for the *LateUpdate* can be seen in figure 5.10.

Lastly we will write our own *Validate()* method that will lock the camera's position to the desired position while we are in the editor.

```
protected override void Start()
{
    Events.LateUpdate += LateUpdate;
}

private void LateUpdate()
{
    // Skip if not attached properly
    if (TargetTransform == null) return;
    // Calculate the desired position (constant offset)
    Vector3 desired = CalculateDesiredPosition();
    // Smooth out the camera movement
    Vector3 smoothPosition = Vector3.Lerp(
        Transform.Position, desired,
        smoothSpeed * Time.Scene.DeltaSeconds);
    Transform.Position = smoothPosition;
    // Set a constant rotation for the camera
    Transform.Rotation = Quaternion.LookAtRH(Vector3.Zero,
                             Vector3.ForwardRH, Vector3.Up);
}
```

Figure 5.10: Using the Events.LateUpdate inside our camera

5.7. Controlling the ship

Next we will create a component *PlayerController* which will react to the key presses and control the player's ship. This component will need a reference to the *RigidBody* component in the same game object so it can apply forces to it. It will also need a reference to the *ParticleGenerator* inside the children game object which we called "Thruster". Figure 5.11 shows the code inside the *Start()* method that does exactly that.

```
rigidBody = C<RigidBody>();

thruster = GameObject.Children.FirstOrDefault(
 x => x.Name == "Thruster")?.TryGetComponent<ParticleGenerator>();
...
Events.Update += Update;
```

Figure 5.11: PlayerController.Start()

Inside the *Update()* method, we will check for the key presses and control the ship using them by applying the proper forces, torques, and controlling the thruster particle generator. Extract from the *Update()* method is shown in figure 5.12.

```
// Go forward if we are pressing the “forward” button
if (Input.IsKeyDown("forward"))
{
    rigidBody.ApplyForce(Transform.Forward * ForwardSpeed);
    thruster.EmitPerSecond = 300;
}
else
    thruster.EmitPerSecond = 0;
...
// Roll to the left if we are pressing the “rollLeft” button
if (Input.IsKeyDown("rollLeft"))
    rigidBody.ApplyTorque(Transform.Back * RollSpeed);
...
// Limit the speed
var vel = rigidBody.BulletRigidBody.LinearVelocity.ToSharpDX();
var len = vel.Length();
if (len > MaxSpeed)
    rigidBody.BulletRigidBody.LinearVelocity *= MaxSpeed / len;
```

Figure 5.12: Controlling the ship inside the PlayerComponent.Update()

With the camera and player controller finished, we can try the scripts out. First we build the game project and then we reload the scripts by selecting *Other → Reload scripts* in the menu. All we need to do is to add the *ShipCamera* under the “Camera” game object and *PlayerController* under the “Player” game object. When we press *Play*, we should be able to control our ship and fly around.

5.8. Health

Next we will need some simple way to damage and destroy objects. We create a component class *Damageable* containing a field *Health : int*. We will also add events *Damaged* and *Dead*, corresponding to receiving any damage and the health getting to zero. Lastly we add a *Despawn : bool* field, which if set to true, will destroy the game object when health goes down to zero.

We will add a method *Damage(damage : int)* that will subtract the damage from the remaining health. Whenever the component receives damage, it will trigger the event *Damaged*, and if the health goes down to zero, it will trigger the event *Dead*.

We will add this component to the “Player” game object and set the health to 150. We will later set asteroids to deal 50 damage which will mean the player can effectively withstand 2 hits and dies with the third.

5.9. Damage sources

Next we will add two component classes which will deal damage on collision. We will call them *Asteroid* and *Shot*. *Asteroid* will only deal damage to the player while *Shot* will deal (lethal) damage to the asteroids.

In order for the classes to deal damage, then try to find a *ColliderComponent* on the same game object and register a handler for the *Collision* event. When a collision occurs, the components will simply check if the name of the game object is “Player” and decide whether they should damage it based on that. Lastly, the *Asteroid* component will also register a handler for the *Dead* event so it can explode as it gets shot down. Figure 5.13 shows the *Start()* method of the *Asteroid* component.

```
protected override void Start()
{
    // try to find the collider component
    var collider = GameObject.Components.FirstOrDefault(
        x => x is ColliderComponent) as ColliderComponent;
    // and subscribe to the Collision event if we found it
    if (collider != null)
        collider.Collision += Collision;

    // subscribe to the Dead event of Damageable component.
    // if the game object doesn’t have the Damageable component,
    // this will create it
    C<Damageable>().Dead += Dead;
}
```

Figure 5.13: Start() method of the Asteroid component

5.10. Lasers

Now we want our ship to shoot lasers.

First we will prototype a game object for the shot. We give it a mesh renderer with some cylindrical mesh and create a new *SimpleMaterial* called *red* which will use the *Emission* property to shine in red. For collisions we will set up a simple *BoxCollider* matching the size of the laser. The prototyped laser is shown on the picture 5.14.

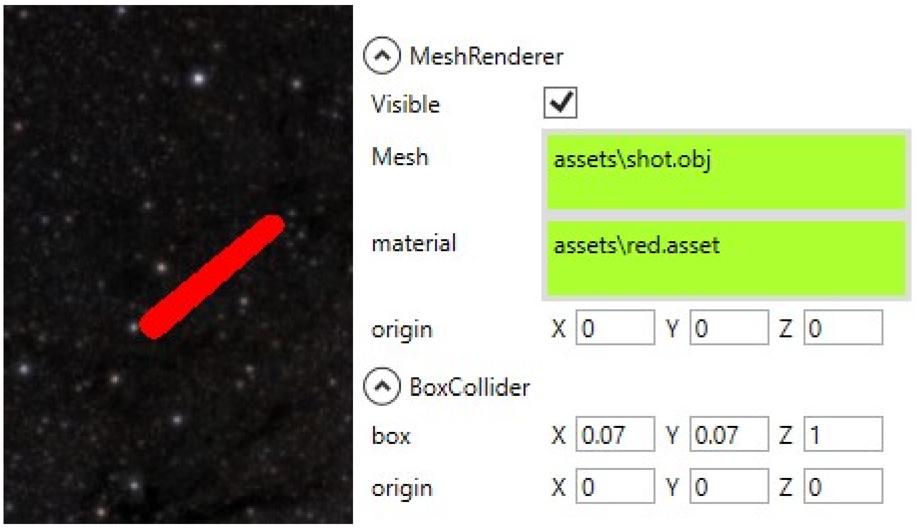

MeshRenderer
Visible ✓
Mesh assets\shot.obj
material assets\red.asset
origin X 0 Y 0 Z 0
BoxCollider
box X 0.07 Y 0.07 Z 1
origin X 0 Y 0 Z 0

Figure 5.14: Laser prototype

Now to shoot the lasers, we will add a component class *Cannon*. This cannon will have a method *Shoot()* that will generate the laser shot and set it flying forward. This *Shoot()* method will simply instantiate the shot with the properties we have prototyped plus the *Shot* component and a *RigidBody* component with the initial speed in the forward direction. Lastly we will create a new component called *DistanceDespawn* which will destroy a component if it gets too far away from the player, otherwise the number of components would keep constantly increasing.

As for the *Cannon* component, we add it to the respective children game objects under the "Player" game object. We will then locate those *Cannon* components inside the *PlayerController* so that we can call the *Shoot()* method on them. We will use the code in figure 5.15 to get the cannons and then we will add the code in figure 5.16 to the *Update()* method to shoot from them.

```
leftCannon = GameObject.Children.FirstOrDefault(
 x => x.Name == "LeftCannon")?.TryGetComponent<Cannon>();
rightCannon = GameObject.Children.FirstOrDefault(
 x => x.Name == "RightCannon")?.TryGetComponent<Cannon>();
```

Figure 5.15: Finding cannons inside of the Start() method

```
if (Input.IsKeyPressed("shoot"))
    if (shot++ % 2 == 0)
        leftCannon?.Shoot();
    else
        rightCannon?.Shoot();
```

Figure 5.16: Shooting from the cannons inside of the Update() method

5.11. Asteroid

Next we will add asteroids. We will create a component class *AsteroidSpawner* that will keep sending asteroids towards the player. This *AsteroidSpawner* will have the following parameters used to control the spawning:

- *SpawnFrequency : float* – spawn frequency of asteroids batches per second
- *SpawnAmount : int* – how many asteroids should be spawned each batch
- *ForwardSpeed* and *SpeedSpread : float* – each asteroid will fly with the speed of *ForwardSpeed* ± *SpeedSpread*
- *SpawnDistance* and *SpawnRange : float* – the asteroids will spawn *SpawnDistance* far away from the player in a square of length *SpawnRange*
- *MinScale* and *MaxScale : float* – the size of the asteroids
- *DespawnDistance : float* – how far away from the player the asteroid has to be to despawn
- *Inaccuracy : float* – asteroids will fly towards a sphere centered around player with this value as a radius

We subscribe to the *Update* event so that we can keep spawning the asteroids.

For the asteroids that will get spawned, we will use the same strategy as with the shot. We will first prototype the asteroid in the editor and then copy the properties in our generator. In particular, the components that will make up each asteroid are: *MeshRenderer, RigidBody, Collider, Damageable, Asteroid,* and *DistanceDespawn*.

5.12. User Interface

Our game is almost finished, we just need to create a user interface. First we add a new *UserControl* in the visual studio and name it *UserInterface*. We then change the base class for the component from *UserControl* to *GridComponent*. We have to do this both in the .xaml and the .xaml.cs file. Note that to set the *GridComponent* as a base class in the .xaml file, we have to define the *Creati* namespace. How to do this is shown in the figure 5.17.

```
// UserInterface.xaml.cs
public partial class UserInterface : GridComponent {...}

// UserInterface.xaml
<creati:GridComponent x:Class="game.UserInterface"
             xmlns:creati="clr-namespace:Creati;assembly=Creati"
             ...>
```

Figure 5.17: Creating a user interface class

To make it simple, we create a global variable for the number of asteroids and for the health left. In the user interface we add a label and a progress bar for the healthy and a label for the asteroids shot down. We then register an *Update()* method to simply keep these values up to date.

We will also create a second user interface called *NewGame*. This user interface will show the score achieved and a button to restart the game. We can see both of those user interfaces in chapter *5.1. Demo walkthrough*.

5.13. Scenes

When we lose, we would like to switch to a different scene that only shows the *NewGame* user interface. This screen should show up when the player dies, so we react to that event and then switch the scene. Actually, we would like the scene switching to be a little bit delayed. To do that, we create a new component class called *DelayedSceneChange* which will wait a set amount of time and then switch the scene. This class will have two properties:

- *ScenePath : string* – the path to the new scene
- *TimeSeconds : float* – the time in seconds, controlling the delay

Then when the player game object dies, we will create a new game object with a single component of *DelayedSceneChange*. To change the active scene, we will simply assign our new scene to the property *SceneManager.Active*.

We also want to start a new game when we click the *restart* button in the *NewGame* interface. For this, we can simply register an event handler to the button click like we would in a normal WPF application and make it switch the scene.

To switch the scene, we have to be careful about one thing – if we use *Assets.Get<Scene>("assets\\startup.scene")*, we will get the scene with the already destroyed ship. We want to load this scene again from the scene file. For this, we can use the method *Assets.LoadAgain<T>(...)* used in figure 5.18.

```
SceneManager.Active =
    Assets.LoadAgain<Scene>(@"assets\startup.scene");
```

Figure 5.18: Reloading the startup scene

5.14. Explosions

As a last detail, it would be nice if the asteroids and the ship exploded after death. We can do this by reacting to the *Dead* event and then creating a new game object in the place of the dead game object. This new game object will have a single component *ParticleGenerator*. We will make this *ParticleGenerator* generate all the particles at once by calling the *EmitRandomParticles(count : int)* on it. After the particles disappear, we need to clean up the game object. We will create a simple component class *TimeToLive*, which will destroy the game object after a set amount of time.

5.15. Finishing

With all of that, we have the demo finished. We can now play it in the debug mode by pressing *Play* or export it and play it in the game launcher.

6. Conclusion

In chapter *1.2. Goals* we have set goals for our thesis. We will now evaluate the fulfillment of those goals:

1. Create a fully extendable game engine core with the following features:
    a. Extendable game object system – the game engine supports the creation of custom components capable of doing the same things as the integrated components
    b. Ability to create new systems – we are able to use the same tools as integrated systems. The game engine gives us tools to create new systems such as the classes *SceneStatic, UnorderedComponentList,* and *CreatiEvent*. The game engine also contains a game loop that we can modify to integrate our system directly into the game engine.
    c. Extendable rendering system – the game engine gives us a number of ways to extend the rendering system. We can create custom materials for meshes, write custom graphical components using custom rendering steps, or even create a whole new rendering pipeline.
    d. Extendable physics system – the physical system included in the game engine is pretty basic. However we are given access to the underlying *Bullet Physics* objects which allows us to extend it rather easily.
    e. User interface system – users can create user interfaces using WPF by implementing the *GridComponent*.
    f. Input system – users can react to all keyboard key presses and mouse buttons as well as the mouse movements themselves.
    g. Asset cache supporting custom resource types – class *Assets* can load various types of resources. Users can extend classes *AssetLoader* and *AssetDefault* to teach the game engine how to load any possible type.
    h. Saving and loading of scenes – the game engine implements a custom serializer that can serialize all objects while cooperating with the asset cache to avoid unnecessarily serializing objects loaded using asset cache
2. Create an editor with sufficient features for basic game development allowing us to:
    a. Create and edit scenes, game objects, and components – the editor can create and edit scenes, edit the game object hierarchy, add and remove components inside game objects, and edit the values inside the components in the inspector.
    b. Debug the game and inspect the properties of our game objects and components while debugging – the editor can debug the game by pressing the play button while keeping the ability to inspect and edit game objects and components.
    c. Easily define a custom user interface for editing components – we can define a custom editor for any data type. We can easily choose which editor should be used for our fields and properties, or even generate a whole interface from scratch. Our components can also react to the changes made in the editor either through the *Validate()* method or by defining custom editor buttons.
    d. Link resources and game objects – we can drag and drop game objects and assets into members of components of the appropriate type to easily link them together.

e. Create custom resources from classes – we can mark class with the attribute *[AssetClass]* and then create instances of that resource inside the editor and assign that resource to various components
f. Write custom scripts that can extend any part of the game engine – the editor creates a visual studio together with the creati project and allows us to write custom classes, components, editors, asset loaders, and more. We can then compile our project and then use the new components and other classes inside of the editor.
g. Export the game as a standalone app – the editor has a menu option *Export* which exports the game created in our editor together with a launcher so it can be launched without the editor.

All of the goals were successfully completed.

Future work

This project only implements the core of a game engine and can benefit a lot from future improvements. Such improvements might include:

- Implementing the CLSL mentioned in chapter *1.1. What is a game engine?*
- Improving the graphical system by adding more components and features, improving the performance, or integrating special graphical editors in the *Creati Editor*.
- Improving or reworking the physical system
- Implementing a sound engine
- Implementing a system for AI
- Adding a support for multiplayer games

7. Bibliography

1 What is a game engine? https://gamescrye.com/blog/what-is-a-game-engine/.
2 Rust programming language. https://www.rust-lang.org/.
3 Lua programming language. https://www.lua.org/.
4 Portal 2. https://store.steampowered.com/app/620/Portal_2/.
5 Multiple inheritance in C++ and the Diamond problem. https://www.freecodecamp.org/news/multiple-inheritance-in-c-and-the-diamond-problem-7c12a9ddbbec/.
6 Jason Gregory. Game Engine Architecture 3rd edition.
7 Cooperative multitasking. https://en.wikipedia.org/wiki/Cooperative_multitasking.
8 Ogre3D. https://www.ogre3d.org/.
9 Urho3D. https://urho3d.io/.
10 SharpDX. http://sharpdx.org/.
11 System.Windows.Forms.Form. https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.form?view=net-5.0.
12 WPF DirectX Interop. https://github.com/microsoft/WPFDXInterop.
13 Brent Owens. Forward rendering vs deferred rendering. https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering—gamedev-12342.
14 Justin Stenning. Direct3D Rendering Cookbook.
15 DeviceContext.Draw. https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11devicecontext-draw.
16 Direct3D Devices. https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-devices-intro.
17 Getting started with the Input-Assembler Stage. https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-input-assembler-stage-getting-started.
18 Direct3D 11 Graphics pipeline. https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline1.
19 Skybox tutorial. https://ogldev.org/www/tutorial25/tutorial25.html.
20 Point lights. https://www.braynzarsoft.net/viewtutorial/q16390-17-point-lights.
21 Phong reflection model. https://en.wikipedia.org/wiki/Phong_reflection_model.
22 System.Windows.Controls.ContentPresenter. https://docs.microsoft.com/en-us/dotnet/api/system.windows.controls.contentpresenter?view=net-5.0.
23 System.Windows.Controls.Grid. https://wpf-tutorial.com/panels/grid/.
24 Bullet Physics. https://pybullet.org/wordpress/.
25 PhysiX. https://developer.nvidia.com/gameworks-physx-overview.
26 Havoc Physics. https://www.havok.com/havok-physics/.
27 Unity. https://unity.com/.
28 Unreal Engine. https://unrealengine.com.
29 DirectX Shader Reflection. https://docs.microsoft.com/en-us/windows/win32/api/d3d11shader/nn-.d3d11shader-id3d11shaderreflection

30 D3DCompiler. http://sharpdx.org/wiki/class-library-api/d3dcompiler/.
31 How to create a texture. https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-resources-textures-create.
32 ID3D11Device::CreateShaderResourceView. https://docs.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11device-createshaderresourceview.
33 Windows Imaging Component. https://docs.microsoft.com/en-us/windows/win32/wic/-wic-about-windows-imaging-codec.
34 AssimpNet. https://www.nuget.org/packages/AssimpNet.
35 XML. https://www.w3.org/standards/xml/core#:~:text=What%20is%20XML%3F,more%20suitable%20for%20Web%20use.
36 JSON. https://www.json.org/json-en.html.
37 Mariana Berga. JSON vs XML. https://www.imaginarycloud.com/blog/json-vs-xml/.
38 Json.NET. https://www.newtonsoft.com/json.
39 System.Text.Json. https://docs.microsoft.com/en-us/dotnet/api/system.text.json?view=net-5.0.
40 System.Window.FrameworkElement. https://docs.microsoft.com/en-us/dotnet/api/system.windows.frameworkelement?view=net-5.0.
41 System.Runtime.Lader.AssemblyLoadContext. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.loader.assemblyloadcontext?view=net-5.0.
42 Jay Strawn. Design patterns by tutorials: MVVM. https://www.raywenderlich.com/34-design-patterns-by-tutorials-mvvm.
43 Using the DataContext. https://wpf-tutorial.com/data-binding/using-the-datacontext/.
44 WPF ICommand in MVVM. https://www.c-sharpcorner.com/UploadFile/e06010/wpf-icommand-in-mvvm/.
45 Assimp file formats. https://en.wikipedia.org/wiki/Open_Asset_Import_Library#:~:text=Assimp%20currently%20supports%2057%20different,functionality%20for%20some%20file%20formats.

8. Attachments

Contents of the attached CD:

- /Sources – contains the visual studio solution containing all the projects
- /Sources/Creati – the game engine project
- /Sources/Creati Editor – the editor project
- /Sources/Game – the launcher project
- /Demo – the demo project built inside the Creati Editor
- /Creati Editor – the release build of the Creati Editor
- LICENCE.txt – file containing the licence
- Thesis.pdf – file containing this thesis
- README.txt – file describing the content of the CD