UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Game Engine for efficient 3D puzzle games

Author:
Nazariy BACHYNSKY

Supervisor:
Andrew IVASIV

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2021

Declaration of Authorship

I, Nazariy BACHYNSKY, declare that this thesis titled, "Game Engine for efficient 3D puzzle games" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ______________________________

Date: ______________________________

"If you optimize everything, you will always be unhappy."

Donald Knuth

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Game Engine for efficient 3D puzzle games

by Nazariy BACHYNSKY

Abstract

Programming 3D graphics is a pretty challenging task. However, such a technology is widely helpful in different areas, for example, architecture prototyping, digital showrooms, or creating movies. Furthermore, one of the most popular usage cases is the video game industry. As it happens when some technology is widely used, there is much software that already implements this technology. The 3D rendering software targeted for creating games is called *game engines*. However, modern game engines also can be used for other purposes. Such flexibility gives an overwhelming functionality for those who do not want to create something huge. We created a light-weighted game engine that does not require much computer resources (powerful CPU and GPU) to run and still is capable of creating 3D games.

The source code can be found here: `https://github.com/nazariyb/FRTEngine`.

And the freshest builds are placed here: `https://github.com/nazariyb/FRTEngine/releases`.

Acknowledgements

I would like to express sincere gratitude to my supervisor Andrew IVASIV, who helped me finish this writing-a-thesis journey. His priceless advice helped me make the right decisions while researching such a complex topic like game engines.

I am beyond measure grateful to Rostyslav Hryniv and Oleg Farenyuk for their courses, respectively, Linear Algebra and Architecture of Computer Systems. They made an immeasurable impact on my work.

I am obliged to Ukrainian Catholic University, especially the Faculty of Applied Sciences for encouraging me to dream big and always do my best, and for infusing my passion for computer science.

Contents

Listings

List of Figures

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
RAM	**R**andom-**A**ccess **M**emory

Glossary

AAA (games) Games which require high budget for development. It can be tens of millions or more. 1

Game mechanics Game mechanics is a combination of restrictions and things player can do in a game. 1

Gameplay The way players interact with a game and what they feel during that. Sometimes is used as synonymous to "game experience". 1

Puzzle video game A game which gameplay is based on solving puzzles. 1

Chapter 1

Introduction

1.1 Motivation

Nowadays, game engines take a prominent place in the computer technologies world. Even though they are called to be *game* engines, it is only for historical reasons. This type of software is also widely used in such fields as architecture, automotive and transportation, broadcast and live events, film and television, training and simulation, etcetera. Still, the game industry entirely covers our field of interest. For comparison, the movie industry surpassed $100 billion in revenues for the first time in history, with earnings reaching $101 billion in 2019[1], while games overtake this number in 2017 by earning $108.9 billions[2] and in 2019, its revenue hit $150 billion[3], and it is instantly growing.

1.2 Problem

Since most game engines are made to cover as many as possible needs (as for different genres of games and other fields, too), they become too overwhelmed when developing a simple game. In this work, we refer to a *simple* game that does not need a tone of technologies and features commonly included in game engines. Those are realistic physics and lighting, landscape building, super cool (realistic or just very detailed) sky/animations/clothing/etcetera, advanced AI system, so on. Even if a game developer does not use those features explicitly, they can be partly involved by some stuff that a person is using. Moreover, such tools make development tools heavier, which means both a programmer and a player need a more powerful computer for running them properly.

1.3 Context

The best example of simple games is puzzle games. Such games do not require much content, high levels, or an extensive library for simulating physics and have an accent on mechanics and exciting riddles. A great example of puzzle games is Tetris. It demonstrates how the quality (of the idea) beats the quantity (of mechanics and content) and confidently competes with AAA games. Different Tetris titles take two entries in the Top-10 most sold game titles over history[4]. The Tetris franchise

[1]https://variety.com/2020/film/news/global-entertainment-industry-surpasses-100-billion-for-the-first-time-ever-1203529990/

[2]https://vgsales.fandom.com/wiki/Video_game_industry

[3]https://www.marketwatch.com/story/videogames-are-a-bigger-industry-than-sports-and-movies-combined-thanks-to-the-pandemic-11608654990

[4]https://en.wikipedia.org/wiki/List_of_best-selling_video_games

is the second most sold game franchise ever, with 495 million downloaded/sold copies[5]. Therefore, we will use it as the primary reference for our solution and for a demonstration of the former.

1.4 Proposed solution

We suggest making our game engine that satisfies minimal requirements for puzzle games. As a result, it will not contain any unnecessary code, hence will not use computer resources that it does not need.

1.5 Goals

Even game engines for simple titles still might have many things to be done. Shortly our task is to have the main game mechanics running. Hence we define two goals:

1. The first one is to implement basic game engine functionality:
   - *Low-level*: Windows application, Input handling
   - *Utilities*: Event system, Logging, Time helper
   - *Rendering*: Camera, Graphics resources, Rendering pipeline
   - *Game architecture*: Game objects, Game world
   - *Visual effects*: Shaders, Dynamic lighting
2. To demonstrate how does the engine work and how it can be used, we will implement basic Tetris mechanics:
   - Tetris board
   - Tetrimino (aka tetromino)
   - Automatically falling + full-controlable[6] tetrominoes
   - Clearing of lines
   - Leveling

Expected result

It is not considered to make a copy of Tetris; we want to make a game with a ruleset based on the game Tetris. More information about it can be found here: `https://en.wikipedia.org/wiki/Tetris`.

1.6 Chosen technologies

As the base language of an application, we have chosen C++17. It is a compelling and efficient language, which has been used for decades. The latest fully supported version grants new features that are convenient in usage and still efficient. We will exploit GPU using DirectX12 - one of the leading libraries on the market. It provides an API for high-performance utilization of GPU and allows to make games that run on Windows (one of the most popular platforms for gaming) and Xbox. Based on this list, for compilation, we will use Microsoft Visual Studio Compiler 2019.

[5]`https://en.wikipedia.org/wiki/List_of_best-selling_video_game_franchises`
[6]`https://tetris.fandom.com/wiki/Tetromino`

Chapter 2

Game Engines

2.1 What is a game engine?

Game engines are often confused with a development environment that contains a user interface for creating and editing game worlds (for example, Unity Editor or Unreal Editor). However, according to Wikipedia, a game engine is *conceptually the core software necessary for a game program to run correctly*. Therefore, technically, a game engine could be (and in most cases, it is) a bunch of dynamic libraries. Also, from the definition, we see no specified list of features that should be present in a game engine - it depends on the needs of the specific game. In Figure 2.1 it is shown a diagram of possible game engine architecture. There are only a few engines that probably have all mentioned parts. However, there are no engines that have something that is not listed here. More information about the architecture of (massive) game engines can be found in Section 1.6 of Gregory, 2018

2.2 Similar solutions

2.2.1 Unreal Engine

Unreal Engine 4 is a potent tool for prototyping and creating games. It started its journey in 1998 and continues improving.

One of its powerful tools is `Blueprints`. `Blueprints` is a system for visual programming which allows doing pretty much anything that can be done in code. Visual programming is easy and fast in development. However it has a cost. It is less efficient than a code, although this is not noticeable for relatively small games. Because of its nature, it is hard to keep visual "code" clean. When the project grows and logic becomes more complex, the "code" turns into a mess.

Besides that, UE4 (Unreal Engine 4) also has many convenient and powerful tools for rig and timeline animations, creating a user interface, editing the game world, creating and editing materials. Its powerful rendering part is why UE4 can be used even for creating movies that look like they were filmed in the traditional way (for example, the series The Mandalorian). Its optimized networking code makes games, created with Unreal, capable of handling online events for millions of players. For instance, during Travis Scott's Astronomical live event in Fortnite, there were 12.3 million concurrent players online[1].

When the game is made with UE4, it is programmed in C++.

[1] https://www.statista.com/statistics/1097635/fortnite-travis-scott-players/

2.2.2 Unity

Unity allows deploying games on mobiles, consoles, desktop computers, TV boxes, web, and virtual reality systems. It has a wide variety of tools and plugins. Among them, it is possible to apply animation created for one object to other. Thanks to its component-based architecture, it is effortless to use. There is no need to care about a vast hierarchy; if something should be added to the game, just create a script, sprite, mesh, or another component and drag-and-drop it on `GameObject`.

Even though it has a bunch of performance issues and glitches, it is also actively growing, as well as the number of noticeable games made with Unity[2]. Due to its simplicity and amount of courses and documentations about Unity, many people start their career in the game industry with this game engine. Furthermore, much of them do not leave it. In 2019 Unity had one and half million monthly active creators[3].

Unity uses C# as a scripting API. This language is pretty easy to learn and provides high performance.

2.2.3 Godot

Godot is a new, growing game engine that perfectly fits for 2D games with pixel graphics. Although it still can be used for 3D, it does not provide a good enough toolset for it and is not recommended for this yet.

2.2.4 Other engines

We will briefly review other game engines, which for one or another reason, do not need much attention in terms of our project.

The Quake family of engines

id Software made the first 3D first-person shooter. It is *Castle Wolfenstein 3D* (1992). It is considered that this company was also the first who started reusing code when creating games. Moreover, this reusable part of code later started to be called a *game engine*. Based on id Software's engine, the following games were made: *Doom*, *Quake*, *Quake II*, *Quake III*, and some other companies' games. Even Valve's *Source* has roots in it.

Source Engine

Valve's Source can rival Unreal Engine 4 in terms of the tool set and graphics. However, it is oriented for creating FPS games, such as the *Half-Life* series.

DICE's Frostbite

This engine has various powerful toolsets and is capable of creating games of different genres, for example, *Mass Effect*, *Battlefield*, *Need For Speed*, *Star Wars Battlefront II* and others. However, it cannot be used outside EA because it is a proprietary engine.

[2] https://en.wikipedia.org/wiki/List_of_Unity_games

[3] https://venturebeat.com/2020/08/24/unity-files-for-ipo-reveals-163-million-loss-for-2019-and-1-5-million-monthly-users/

Rockstar Advanced Game Engine (RAGE)

Grand Theft Auto V and *Red Dead Redemption*, alongside many other games, are based on this engine. It is also capable of creating cross-platform games.

CRYENGINE

The latest version of this engine allows making games targeting almost all popular platforms and provides s powerful suite of tools and high-quality real-time graphics.

Sony's PhyreEngine

Any licensed Sony developer gets access to this engine, which supports developing only for Sony's consoles.

Microsoft's XNA Game Studio

This game engine was based on C# language and aimed at encouraging players to create their games and share them with others. However, XNA is no longer supported since 2014.

GameMaker Studio

Game Maker is costly, though powerful, game engine for 2D games.

Construct

Although it is not as expensive as GameMaker, it is not as powerful as the latter. Moreover, it still aimed for 2D games.

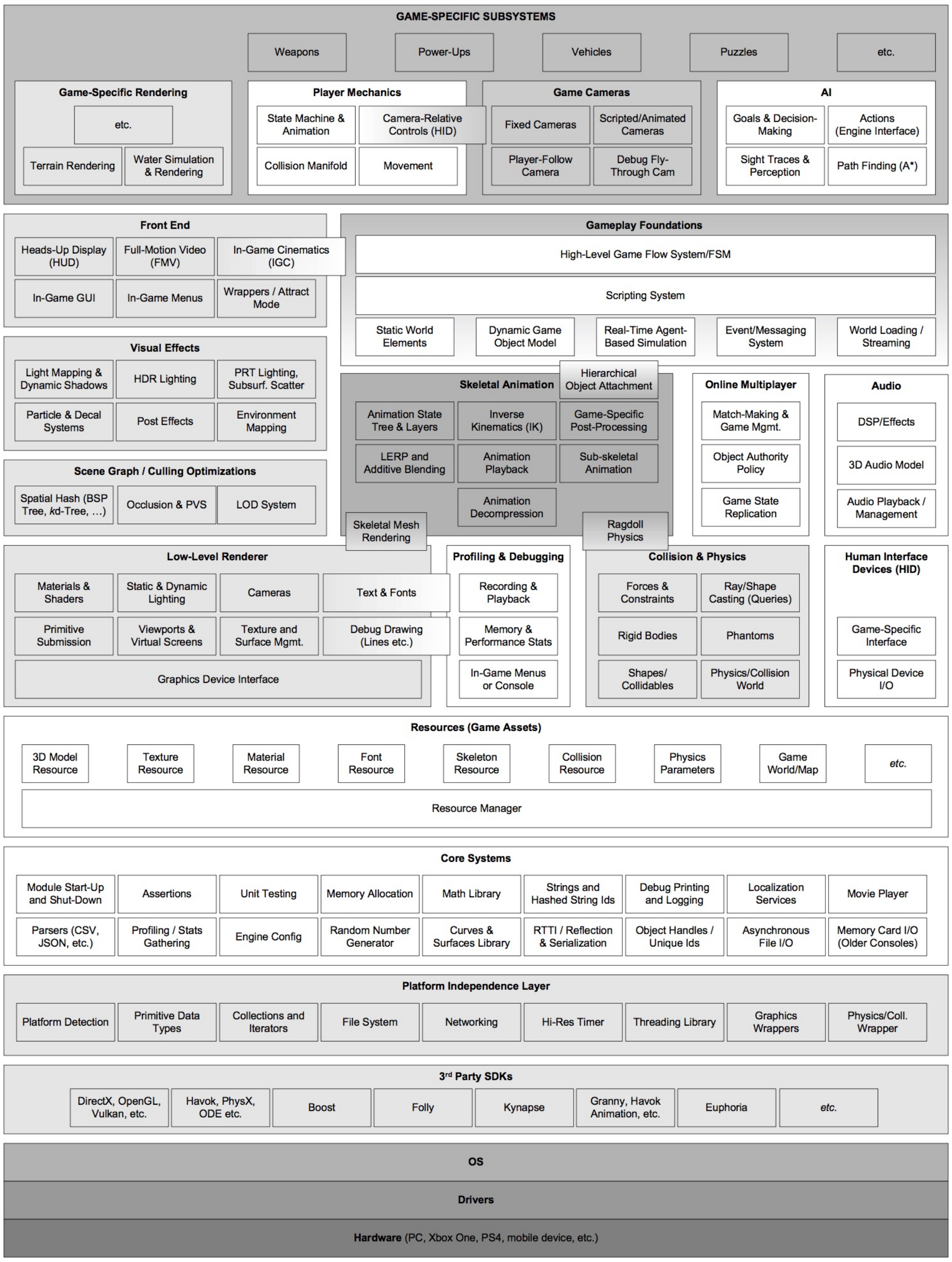


FIGURE 2.1: Runtime game engine architecture
Source: Gregory, 2018

Chapter 3

Technologies

3.1 Overview

Since we want the software to be as efficient as possible, we will use native SDKs and not cross-platform. For idea demonstration, one platform will be enough. Our choice is Windows because it is the most popular platform for gaming. Moreover, if needed, code written for Windows can be extended to running on Xbox without losing any performance. A game is just an application. The most efficient and light-weighted way to make an application on Windows is to use Windows API. The native solution for graphics on the Windows platform is DirectX. We do not have any restrictions, so we will use the latest version, which is 12.

3.1.1 Windows API (WinAPI)

WinAPI provides access to several platform implementations which interact with OS directly. We are going to use it to create and set up a window of an application. WinAPI also grants us the possibility to handle any events by processing messages coming in an endless loop. We are going to use them to handle mouse and keyboard input.

3.1.2 DirectX 12

Microsoft DirectX and Windows API are a collection of APIs for handling tasks related to multimedia, especially game programming[1]. We are going to need two of them: `Direct3D` and `DirectXMath`.

Direct3D

`Direct3D` is the graphics API at the heart of DirectX. It provides very convenient and effective features for interacting with a graphics processing unit. Some of them are `command queues and lists`, `descriptor tables`, and `pipeline state objects`[2]. We will provide more details later, with examples of usage.

DirectXMath

`DirectXMath` provides convenient structures to work with vectors and matrices. Nevertheless, what is great about this library, that it uses the Intel (AMD chips support it too, though) SSE2 (Streaming SIMD (Single Instruction Multiple Data)

[1] https://en.wikipedia.org/wiki/DirectX

[2] https://docs.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12-

Extensions 2) instruction set. SIMD registers are 128-bits wide, allowing SIMD instructions to operate on four 32-bit `floats` or `ints` at once. Using these instructions significantly increases the efficiency of mathematical operations on vectors and matrices, vital in such high-performance and high mathematically dependent software as video games.

3.2 The rendering pipeline

The rendering pipeline is the sequence of steps committed by GPU to draw a 2D picture on screen given geometry description of the 3D world.

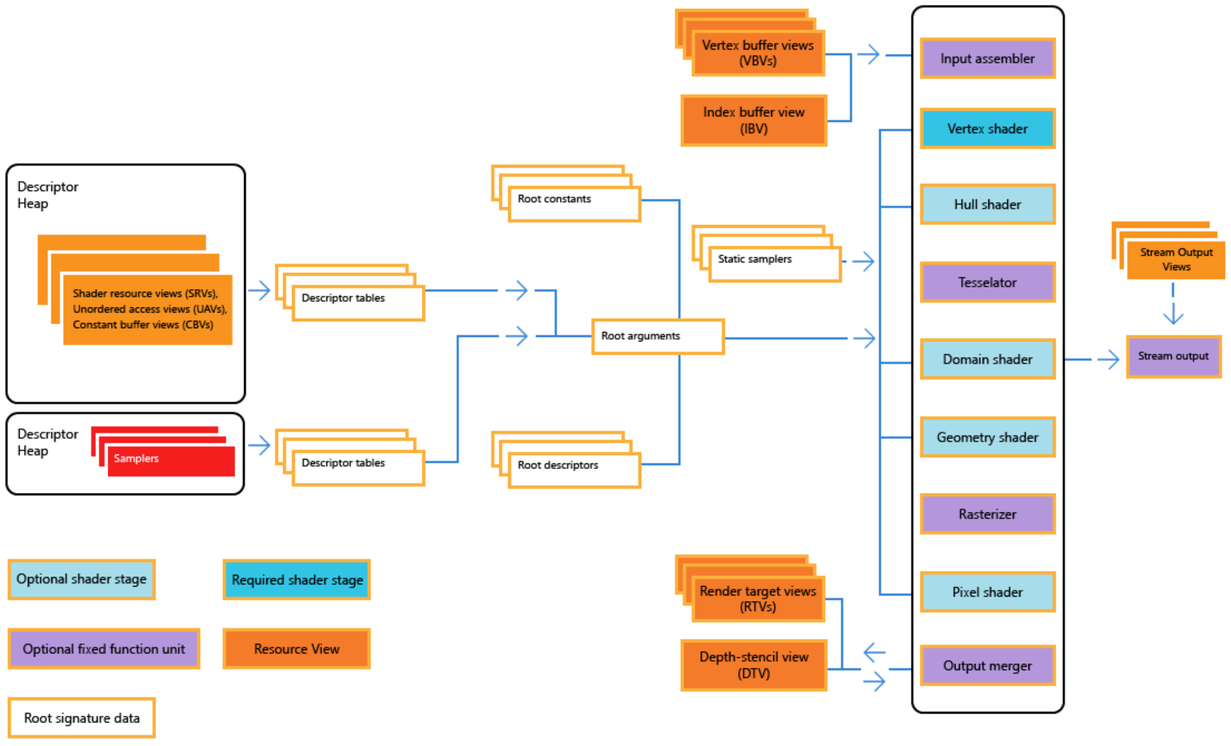


FIGURE 3.1: The Direct3D 12 graphics pipeline and state

In the right part of the Figure 3.1 are listed rendering (graphics) pipeline stages. We are going to discuss them in this section. *Descriptors*, *Descriptor heaps*, *Descriptor tables*, *Root*, and other resources are perfectly explained in Section *Direct3D 12 Programming Guide > Resource Binding in Direct3D 12* of *Direct3D 12 graphics*. *Render target views* and *Depth-stencil view* are covered, respectively, in Section 4.3.7 and Section 4.3.8 of Luna, 2016

3.2.1 The input assembler stage

The goal of the input assembler stage is to read geometric data (vertices, indices) from memory and assemble it into geometric primitives (triangles, lines) in GPU-understandable format.

Vertices

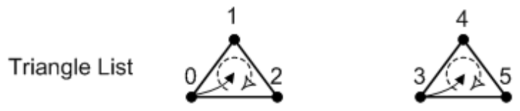
In `Direct3D`, vertices can be considered as an extended version of vertices of geometric primitives. They still can store the spatial location of points where edges of figure meet. However, they also can contain any additional data we need. For our needs, we define the following vertex structure:

```
struct Vertex
{
    DirectX::XMFLOAT3 position;
    DirectX::XMFLOAT3 normal;
    DirectX::XMFLOAT2 uv;
};
```

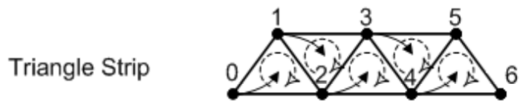
XMFLOAT2 and XMFLOAT3 from namespace DirectX are just structures for representing mathematical 2D and 3D vectors correspondingly. Field position represents the spatial location of a vertex in 3D space. We will discuss normals and uvs later. We can initialize and store vertices data in any way that is convenient for us. And then just give its address to Input Assembler.

Topology

Since we can define almost whatever data format we want to use in code and pass to IA (Input assembler) contiguous data, we have to specify how to read it. Let us start with a topology type. For example, in case we want IA to transform every three vertices into a triangle, we would set Triangle List (Figure 3.2a) topology. We can use Triangle Strip (Figure 3.2b) topology to make IA consider each next triangle shared one side with a previous one. That is, vertices 0, 1, 2 build up a triangle, the next triangle consists of vertices 1, 2, 3 and so on.



(A) Triangle list topology



(B) Triangle strip topology

FIGURE 3.2: Examples of possible primitive topologies.
Dash circled arrow shows an order of vertices for assembling. A short solid arrow indicates the first vertex of a triangle. Source: *Direct3D 11 graphics*

Indices

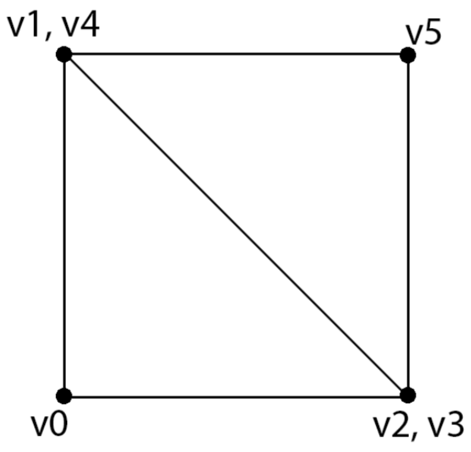
Before continuing, let us clarify why we need those triangles (often referred to as polygons) at all. It is the most efficient way for GPU to work with points, lines, or triangles. We are not using points and lines in our project not to discuss them, but the same logic we use for triangles can be applied to lines and points. The graphics processing unit uses those primitives to build complex solid 3D objects. Vertices and indices are raw materials for this. Say, we have the following vertices:

```
// for sake of example, ignore normals and uvs,
// and consider its to be in two dimensional space
using Vertex = DirectX::XMFLOAT2;

Vertex v0 = { 0.0f, 0.0f };
Vertex v1 = { 0.0f, 1.0f };
Vertex v2 = { 1.0f, 0.0f };

Vertex v3 = { 1.0f, 0.0f };
Vertex v4 = { 0.0f, 1.0f };
Vertex v5 = { 1.0f, 1.0f };
```

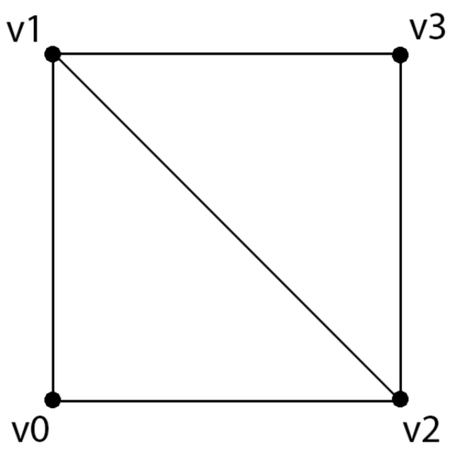
Here, `v0`, `v1`, `v2` are positions of one triangle and `v[3-5]` of another. It would be OK, however, as we see, `v1` and `v2` represent the same geometrical points as `v3` and `v4` (see Figure 3.3a). That is, these triangles are two parts of the same square. Therefore, two of its vertices will be stored and processed twice, not what we want in most cases. In order to use the same vertex multiple times, we have to say IA what vertices to use and where they should be used. This is what `indices` in Direct3D are for.

```
Vertex vertices[4] =
{
    { 0.0f, 0.0f },   // 0
    { 0.0f, 1.0f },   // 1
    { 1.0f, 0.0f },   // 2
    { 1.0f, 1.0f },   // 3
}

UINT indices[6] =
{
    0, 1, 2,   // triangle 1
    1, 2, 3    // triangle 2
}
```

That is, `indices` is an array of numbers that are used literally as indices to subscript an array of `vertices`. Therefore, one triangle will be formed from `vertices[0]`, `vertices[1]` and `vertices[2]`, and the other one from `vertices[1]`, `vertices[2]`, and `vertices[3]` (see Figure 3.3b).



(A) A square built from six `Vertex` objects



(B) A square built from four `Vertex` objects

FIGURE 3.3: Usage of indexing
When not using indices, we use two different `Vertex` objects to define the same vertex. In indexed square, two vertices are associated with the same `Vertex` object.

3.2.2 The vertex shader stage

A `shader` is a code that runs on a GPU. Similar to `C++` programs, shader has a "main" function that serves as an entry point. It takes some data and outputs some data. It also can have access to special registers of shared memory on GPU. Our task here is to define the "main" function, and it will be called for us by the GPU driver. It will be run simultaneously in multiple threads for multiple data since GPU is designed for such operations. There are different shader types for different purposes which work with different data. Asthe `vertex shader stage` name claims, this type of shader works with vertices (see Figure 3.4). Based on data we passed to IA, it assembles primitives and feeds vertices data to the vertex shader.

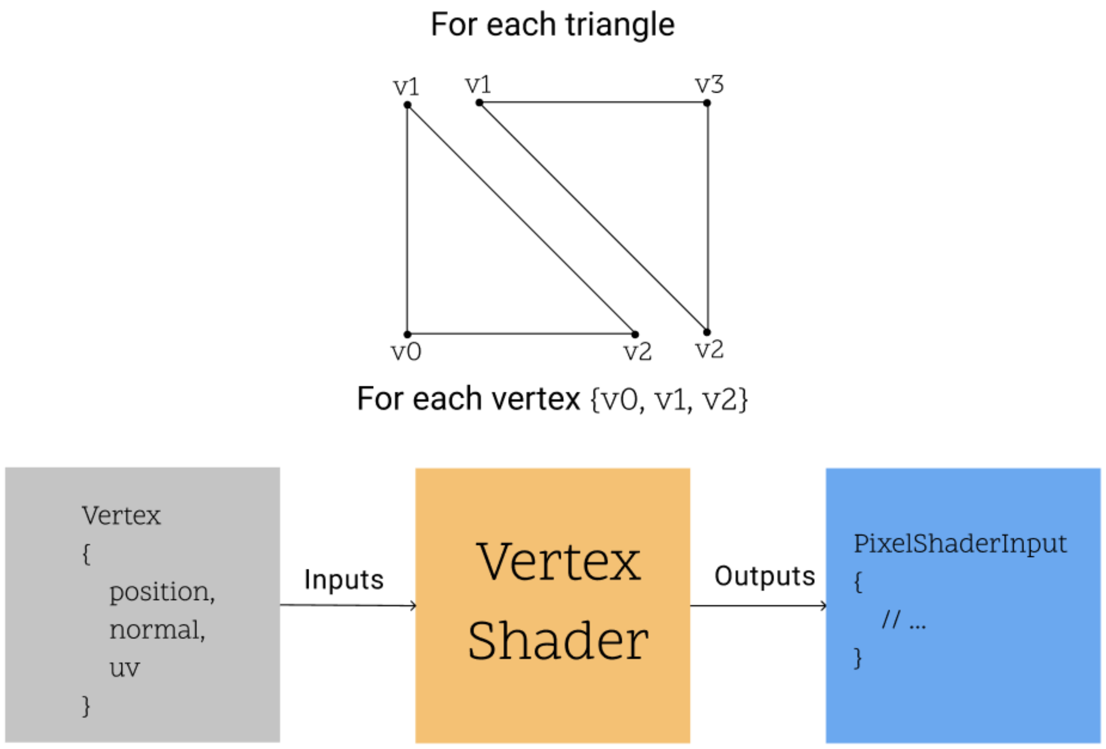
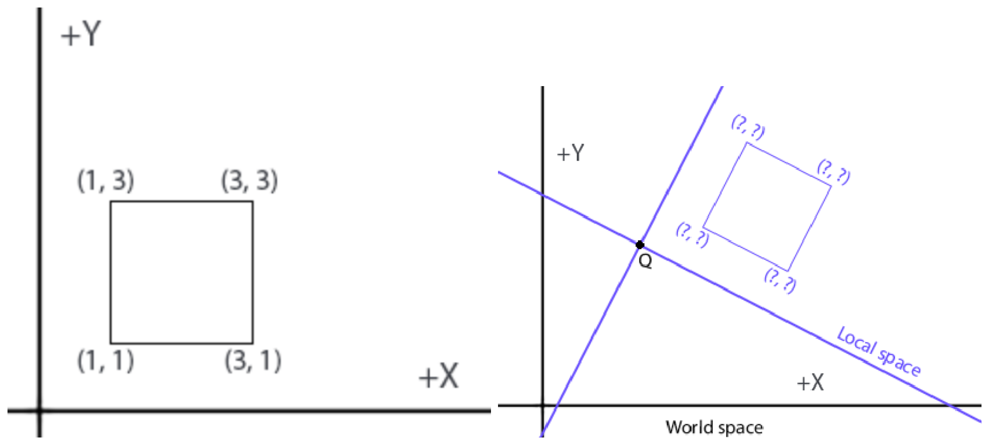

For each triangle
v1
v1
v3
v0
v2
v2
For each vertex {v0, v1, v2}
Vertex { position, normal, uv }
Inputs
Vertex Shader
Outputs
PixelShaderInput { // ... }

FIGURE 3.4: Vertex shader usage scheme
`Vertex shader` code is simultaneously running on multiple cores of GPU and processing vertices of primitives. It takes `Vertex` data as input and return the structure which will be later fed to `pixel shader` as input parameter.

Local space and world space

A game scene or level is just some location that can be mathematically represented by many objects with their coordinates in the scene coordinate system. It is commonly referred to as a world space. However, as we know, 3D objects are not solid items and consist of points (vertices). Thus we have to define the location for every vertex. That means, if we want to draw the same object multiple times in different places, we need to create a distinct object with its vertices, which means duplication of data. Also, we might want to reuse the same object in other scenes. Again, it requires recreating an object with coordinates relative to the new world. The solution is pretty straightforward - to make 3D models in local space. All model points have positions relative to their coordinate space. Each object has its own coordinate space with the origin, for instance, in the object's center.

Thus, when adding an object to the scene, we have to translate its points' positions to world space (Figure 3.5). Since each point (vertex) can be reinterpreted as



(A) (B)

FIGURE 3.5: Local and World spaces
Since we create an object relative to local space (A), later we have to transform it to world space (B) in order to know how to draw

a vector, we have to find the position of this vector relative to world space. Furthermore, suppose we have unit vectors that aim in axis directions. In that case, we can multiply the coordinates of points of the object by the respective vector to get its representation in world space. Mathematically, we can describe as having vector $\mathbf{p}_L = (x, y, z)$ in local space, and we want to find a $\mathbf{p}_W = (x', y', z')$ in world one. $\mathbf{u}_L, \mathbf{v}_L, \mathbf{w}_L$ are unit vectors alongside of axis. That is we can define $\mathbf{p}$ as following:

$$\mathbf{p}_L = x\mathbf{u}_L + y\mathbf{v}_L + z\mathbf{w}_L$$

And, if we know $\mathbf{u}_W, \mathbf{v}_W$ and $\mathbf{w}_W$, then

$$\mathbf{p}_W = x\mathbf{u}_W + y\mathbf{v}_W + z\mathbf{w}_W$$

However, we also have points and not only vectors, so we want to save their locations. Thus we introduce one more variable in our equation - the origin of the local frame. Let us denote it $\mathbf{Q}_L$. Now, we come to the following:

$$\mathbf{p}_L = x\mathbf{u}_L + y\mathbf{v}_L + z\mathbf{w}_L + \mathbf{Q}_L$$

And if we have the coordinates of the origin relative world frame we can always find $\mathbf{p}_W = (x', y', z')$:

$$\mathbf{p}_W = x\mathbf{u}_W + y\mathbf{v}_W + z\mathbf{w}_W + \mathbf{Q}_W$$

See Figure 3.6 for geometry representation of the equation (for 2D case).

It is not very convenient to use two different equations, so we can handle them by one equation by adding one more parameter w. In vectors we set it to 0 so it removes part for changing a location. Setting it to 1, the equation will properly transform points:

$$(x', y', z', w) = x\mathbf{u}_W + y\mathbf{v}_W + z\mathbf{w}_W + w\mathbf{Q}_W$$

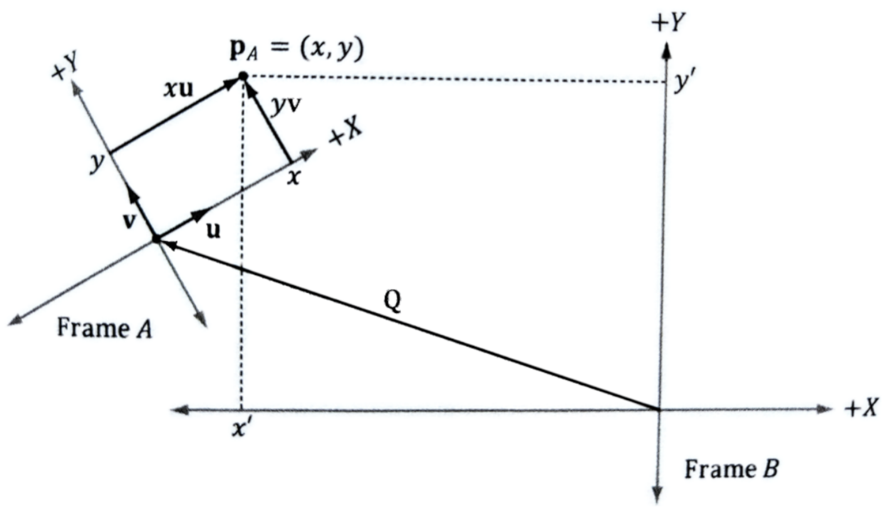


FIGURE 3.6: Change of point's transformation
The geometry of finding the coordinates (x', y') of point $\mathbf{p}$ relative to `Frame B` (world space) knowing coordinates of vectors $\mathbf{u}$ and $\mathbf{v}$ (which aim, respectively, along the $x-$ and $y-$axes of `Frame A` (local space)), and origin $\mathbf{Q}$ relative to world space. Source: Luna, 2016

In fact, we can rewrite it in language of matrices:

$$\begin{bmatrix} x' & y' & z' & w \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

And it gives us a matrix called a world matrix $\mathbf{W}$:

$$\mathbf{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

Although this is a working way, it is not always easy to figure out the coordinates of the local space origin and axes relative to the world space. A more common and convenient way to define a world matrix is to use a sequence of transformation - scaling matrix $\mathbf{S}$ to change the size of an object in the world, then a matrix of rotation $\mathbf{R}$ which defines the orientation of the local space relative to the world one, and to define the origin of the local space relative to the scene the translation matrix $\mathbf{T}$ is used: $\mathbf{W} = \mathbf{SRT}$.

We have all objects in world space, but it is still just a geometrical representation of data. So how do we show it on a 2D screen? We have to use a camera, just like we were shooting a movie. Although our camera is virtual, it can "see" only limited space. That volume is the part of the world that should be rendered and visible to the player. The camera sits at the origin of its own local space called *view space*. The change of coordinate transformation from world space to view space is called the *view transform*, and the corresponding matrix is called *view matrix*.

If the origin, $x-$, $y-$, and $z-$axes of view space with homogeneous coordinates relative to world space, are described, respectively, by $\mathbf{Q}_W = (Q_x, Q_y, Q_z, 1)$,

$\mathbf{u}_W = (u_x, u_y, u_z)$, $\mathbf{v}_W = (v_x, v_y, v_z)$, and $\mathbf{w}_W = (w_x, w_y, w_z)$, then, as we know from 3.2.2, we can change coordinates from local space to world space with the following matrix:

$$\mathbf{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

Nevertheless, in this case, we want reverse effect - change coordinates from world space to local one (view space). We have to use inverse $\mathbf{W}^{-1}$. Since the scale of objects does not change when changing the coordinate system, we can ignore the scale matrix: $\mathbf{W} = \mathbf{RT}$. This form simplifies computation of an inverse:

$$\mathbf{V} = \mathbf{W}^{-1} = (\mathbf{RT})^{-1} = \mathbf{T}^{-1}\mathbf{R}^{-1} = \mathbf{T}^{-1}\mathbf{R}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

When we defined a view matrix, we have to figure out how to construct it. Let $\mathbf{Q}$ be the camera position and let $\mathbf{w}$ be the camera look direction. Both are relative to world space. Furthermore, let $\mathbf{j}$ be the normalized "up" direction. The thing is, sometimes xy-plane can be used as a "ground plane". But more common approach is to use xz-plane for this purpose. We will do so too. Anyway, we need specify it: $\mathbf{j} = (0, 1, 0)$. Now we use cross product to find the vector Which is orthogonal to up direction and $\mathbf{w}$. The latter describes one of axis of camera local space, that is we will find the vector that describes the second "horizontal" axis:

$$\mathbf{u} = \frac{\mathbf{j} \times \mathbf{w}}{||\mathbf{j} \times \mathbf{w}||}$$

Having $\mathbf{u}$ and $\mathbf{w}$, we can find $\mathbf{v}$. Since they are orthogonal units vectors, their cross product is a unit vector too and there is no need to normalize it:

$$\mathbf{v} = \mathbf{u} \times \mathbf{w}$$

Therefore, having camera position, look and up directions, we can derive camera local space which the view matrix will be constructed from.

Projection and homogeneous clip space

Even though our camera is virtual, it still has some restrictions like a real one. It can see only some specific volume, which is described by the frustum. Our task is to project 3D geometry from the frustum to the 2D projection window, parallel to xy-space. The center of projection coincides with a camera position at the origin of the camera's local space. In Section 5.6.3.1 of Luna, 2016 it is described how we can define a frustum having a near plane n, far plane f, vertical field of view angle α, and aspect ratio r. The near and far planes are parallel to the projection window and xy-plane, so they can be described as a value along the z-axis (Figure 3.7 visualizes it). The aspect ratio is given by the ratio of the width of the projection window to its height.

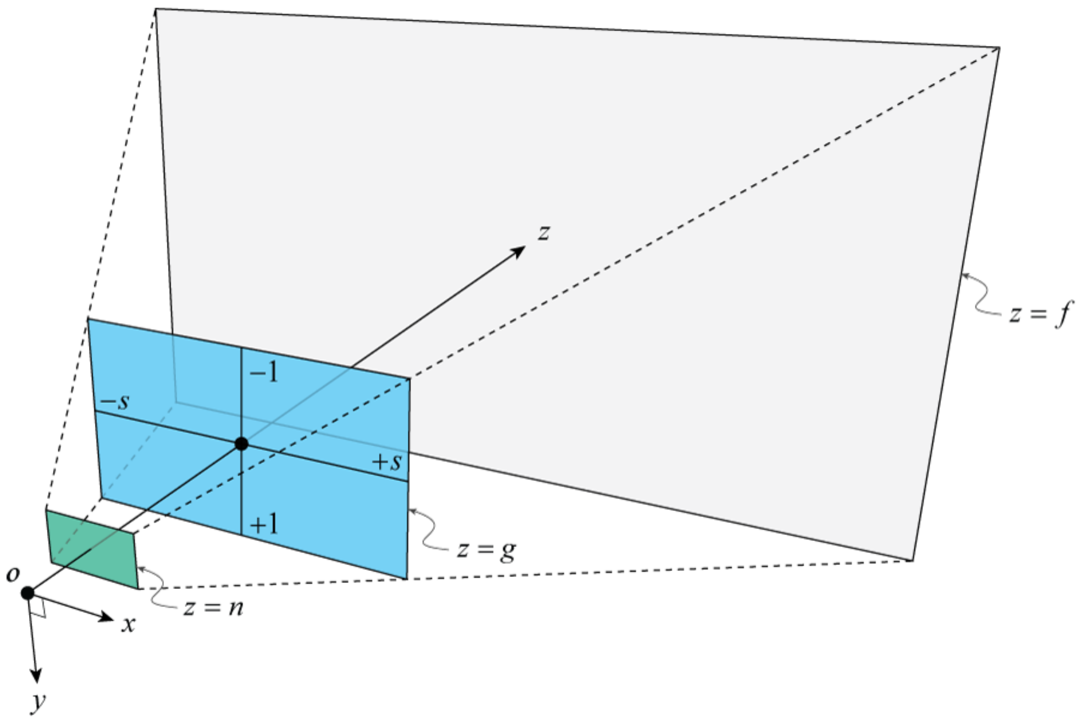


FIGURE 3.7: The camera's view frustum
This is a view frustum shape of a camera whose `look` direction goes alongside $z-$axis, and `up` direction goes in opposite direction to $y-$axis (since $y-$axis goes down in this example). Here, the real resolution does not matter, so the height of `projection` plane is set to 2, and width equals to aspect ratio s ($s = projection\_width/projection\_height$). The `projection` plane itself is in the middle (the blue one), and is placed at distance g from the origin (camera position). The farthest (gray) plane is `far` plane. Everything beyond this plane is not projected onto the `projection` plane (so, is not visible to a player). The nearest to the camera plane (green) is `near` plane. However, this is not always the case, in fact it is more common to encounter the system, where the `near` plane if further from a camera than a `projection` plane. Everything, whose $z-$position is lower than n (`near` plane's z position) is not projected. All three planes are orthogonal to $z-$axis. Source: Lengyel, 2019

A complete explanation of the perspective projection is presented in Section 5.5.1 of Lengyel, 2011. Thus, we will just provide a perspective projection matrix which is used in Direct3D:

$$\begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

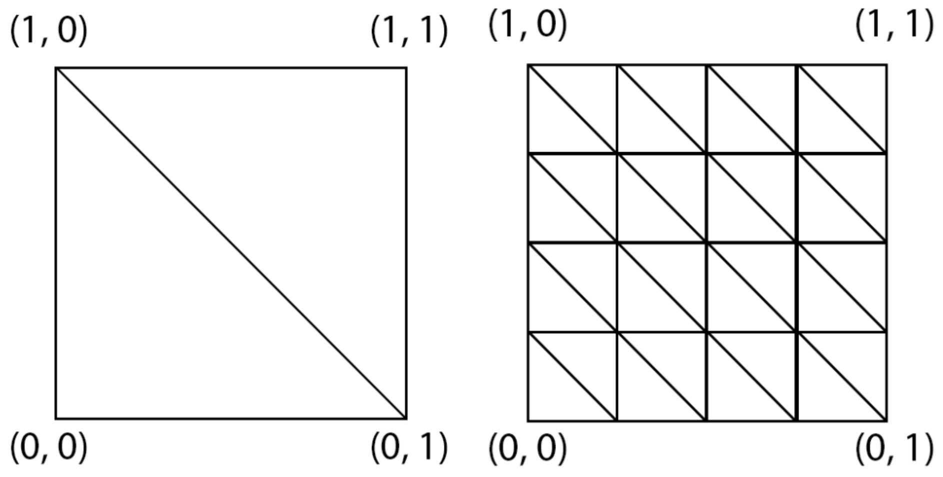
Where n and f are, respectively, near and far planes, w and h are, respectively, width and height of the projection screen.

3.2.3 The geometry shader stage

The geometry shader takes primitives as an input. If we passed the list of triangles to IA, the geometry shader would get three vertices that define a triangle. This shader can destroy, create or tessellate geometry primitives and is used, for instance, to create particles or cloth simulation.

The tessellation

The tessellation means dividing triangles of a mesh into smaller ones. Thanks to this, the mesh becomes more detailed. It is useful, for example, when having complex objects, animations, or creating more details when the camera is near to the object. Figure 3.8 shows an example of tessellation.



(A) Before tessellation (B) After tessellation

FIGURE 3.8: Tessellation
After a tessellation a square still remains the same size, but now it consists of more triangles (polygons). For instance, a waving (vertex-based) animation can be applied to it now.

Stream output

After changing geometry, the data can be written back to memory, from where it will be taken to the top of the pipeline. So, the new geometry will be processed with previous stages and then rendered. This stage is not mandatory and will not be exploited by us.

3.2.4 Clipping

If, after previous operations, some vertices go further than frustum boundaries, they must be clipped. The intersection points will be taken as new vertices for primitives. This stage is completely done automatically by hardware, and we do not have to worry about it.

3.2.5 Triangle traversal

Each triangle is broken into fragments by the rasterizer. Usually, each fragment corresponds to each pixel. This stage also interpolates the vertex data (position, normals, uvs, or others) to generate per-fragment corresponding attributes.

3.2.6 Z-Test

Pixel shaders (which we will talk about in the following subsection) are potentially very computationally expensive. There is no sense in running it on pixels that are further from us than other ones. Newer graphics cards discard such pixels before calling pixel shader on them (older GPUs process pixels first and then discard unnecessary ones).

3.2.7 Pixel shader

A `pixel shader` is sometimes also referred to as a fragment shader. It inputs an interpolated vertices data (such as position, normal, or others) and uses it to calculate the fragment's color. It is very flexible and can have access to multiple textures. The `pixel shader` is used for computing lightings, shadows, reflections, or other various effects.

3.2.8 The output merger stage

This stage takes pixel fragments from pixel shader and runs various tests on them. If tests are passed, it writes a pixel to the back buffer. This stage is also responsible for blending. If an object is semitransparent or translucent, its pixels are mixed with pixels of an object from the back using a special formula, which is not in our lists of subjects for discussion.

Chapter 4

Implementation (FRTEngine)

4.1 Architecture

We decided to call our engine FRTEngine. Every class mentioned in this chapter is nested into a namespace frt. Each class has defined with a macros FRTENGINE_API before its name. This macros unrolls to

```
#ifdef FRTENGINE_EXPORTS
#define FRTENGINE_API __declspec(dllexport)
#else
#define FRTENGINE_API __declspec(dllimport)
#endif
```

FRTENGINE_EXPORTS is pre-defined in FRTEngine project settings. That is, other (game) projects do not have it. Thus, classes defined in base project we mark as ones, that will be exported via dll (dynamic library). When header files of those classes are included in derived projects, we tell a compiler that these classes' implementations will come with dll.

On Figure 4.1 is shown the UML diagram of FRTEngine architecture. Only the most essential class members and methods are listed. Even though it seems a bit messy now, we will explore it step by step logically and understandably. We highly recommend looking at our GitHub repository during or after reading this to understand a concept better.

ITickable

Before we go to the heart of the project, let us discuss a class ITickable which is a base class for almost half of the other classes in the project.

As will be shown in 4.3, some objects have Update() and Render() functions. In fact, almost every object has them (or at least Update()). These methods are called every frame. Update() is considered to calculate a new state of an object and Render() roughly is meant to pass updated data to GPU. However, Render() is quite a wide-meaningful term and it is used only in Graphics. Instead, other objects have a method called PopulateCommandList().

Since most objects are capable of such functionality, we added a simple interface:

LISTING 4.1: Interface ITickable

```
class FRTENGINE_API ITickable
{
public:
    virtual ~ITickable() = default;
    virtual void Update() = 0;
    virtual void PopulateCommandList() = 0;
};
```

4.2 Windows application

Before going to the root of the application - its entry point, and its interaction with OS, let us observe the App class - the class which is created to be an entry point to the *logic* of our application, "the main" class. It (technically, its instance) stores and manages classes (their instances) responsible for fundamental parts of an application. Those are: Window 4.2.1 - it does all work related to communication with OS. It also owns a Graphics 4.4 which is responsible for rendering; GameWorld 4.5.3 - it spawns, destroys and manages GameObjects 4.5.1. Besides other stuff, it also has a static template function (see Listing 4.2) which allows to create an instance of the class derived from App, that is GameApp - the class derived in project of game, for example TetrisApp.

LISTING 4.2: 'Launch' function of App

```
template<class T>
App* App::Launch(HINSTANCE hInstance, HICON icon)
{
    static_assert(std::is_base_of<App, T>::value);
    _instance = new T();
    _instance->Init(hInstance, icon);
    _instance->Run();
    return _instance;
}
```

It is noticeable that we assign the newly created instance of the game app to _instance field of itself: `_instance = new T();`. We do so because our App is considered to have only one instance over entire application, which is pretty obvious, as far as App is associated with the application. Thus, we implemented it with the design pattern Singleton[1] in mind. In App's constructor we create the GameWorld: `world = new GameWorld();`. In Init method the Window (4.2.1) is created, and Time (4.2.5) is initialized:

```
window = new Window(_windowWidth, _windowHeight, _windowName, icon);
Time::Init();
```

`App::Run()` function is pure virtual, so it must be overridden in derived class. The one more important function is `static void Shutdown();`. It deletes App's instance and as result, its destructor is called, which in its turn deletes window and world since they were allocated on a heap. And the last but not least two functions that we want to mention are `Update()` and `Render()`.

They both are evoked successively in a "while-true" loop (see Listing 5.3) in main function. Here are their implementations:

LISTING 4.3: App::Update() and App::Render() implementations

```
void App::Update()
{
    Time::Tick();
    window->GetGraphics().Update();
    world->Update();
}
void App::Render() { window->GetGraphics().Render(); }
```

We are going to discuss mentioned above functions in the following sections.

[1] https://en.wikipedia.org/wiki/Singleton_pattern

4.2.1 Window

Main function (entry point)

As we mentioned, we use Windows API for creating an application itself. First of all, the entry point of such a program differs from standard C++ programs. It has the following signature:

```
int WINAPI WinMain(
    HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow
);
```

Here, WINAPI is typically defined to calling convention __stdcall. Argument hInstance is called a "handle to an instance" and is used by the operating system for executable identification. We will refer to it later. Argument hPrevInstance is present only for backward compatibility and is always zero. The last two arguments we will not use.

For more convenient debugging we have added custom Exception class and throw its instances from anywhere is needed. Then we use MessageBox which is provided by WinAPI to show message about an error occurred and exit the application. As we saw in 4.2, Launch calls function Run which will be overridden in derived class. This function also must contain "while-true" loop of the game to have everything running properly. We will discuss the structure of the loop in Section 5.1 (Listing 5.3).

WinAPI window class

The next step is to register a window class which is nothing like a class in OOP programming languages. Window class is a set of attributes, which includes linking to hInstance. OS associates all these parameters with a name, which we can specify when creating a window. WinAPI allows us to set various window attributes such as title, width, height, location, etc. and a set of special flags that control available window actions and styles. We also associate a window procedure with this window which is basically a function with the following signature:

```
LRESULT CALLBACK MainWndProc
(
    HWND hwnd,          // handle to a window
    UINT uMsg,          // message identifier
    WPARAM wParam,      // first message parameter
    LPARAM lParam      // second message parameter
);
```

In MainWndProc we check message type and send it further if needed. For more convenient handling of those events inside an application we use custom written Events (4.2.2).

For convenience, we created a "wrapper" class for WinAPI Window. Thus, all those initialization will be encapsulated by our Window's methods. We will not describe them in detail. However, besides creating and handling a window (program) on WinAPI level, it also owns Mouse, Keyboard and Graphics (We will talk about them in the following sections). It probably will not be the best solution for big projects. However, it works very well for our project, so this is the case when we should act according to the design principle "Keep it simple, stupid" (KISS). This is not the only place in our project where we apply this principle.

4.2.2 Event system

When something meaningful happens in the game, it is referred to as an *event*. It could be a player pressed a button, one object hit another, an explosion went off, or something else. It is not convenient and makes code messy for an object to know every other object, which should be notified when something happens, and notify them. The solution is to have a special structure, which is commonly called an `Event`. Rarely, `command` or `message` is used. This entity stores information about what happened, for example, what button was pressed. An `event` also has a *list of listeners* that will be notified when needed. The significant advantage of such a system is that the object which generates an event does not need to know what objects want to be notified. They care about *subscribing* to the desired `event`. The *list of listeners* means a list of functions. When `event` is invoked it calls each listener. When an object *subscribes* to an `event` it adds a listener to `event`'s list of listeners. More information can be found in Section 16.8 of Gregory, 2018.

That is, each `event` instance will have its list of listeners, which in our case are functions that must return nothing and take a pointer to `Event`. Such listeners can be added using overloaded operator +=. Example of possible using:

LISTING 4.4: Event usage example

```
struct Object
{
    static Event objectConstructedEvent;
    Object() { objectConstructedEvent.Invoke(); }
};

Event Object::objectConstructedEvent{};
//...

{
    Object::objectConstructedEvent += [](Event*) { std::cout << "One
        more instance of type Object constructed!\n"; };
    Object o1, o2;
}
```

Output:

```
One more instance of type Object constructed!
One more instance of type Object constructed!
```

4.2.3 Input

The place where the events are exploited the most at the moment is the `Input system`. As we mentioned at the beginning of 4.2.1, `WinAPI` sends us system events (messages) for everything. And in our `Window` we process those messages. Then based on message's type we can know whether it is an input event. There are a bunch of predefined types in `WinUser.h` (see Listing 4.5 for examples) (from `Windows SDK`). If message type matches one of these types, we create and delegate a special `MouseEvent` or `KeyboardEvent`.

LISTING 4.5: Example of input event types

```
#define WM_KEYDOWN                0x0100
#define WM_KEYUP                  0x0101
#define WM_CHAR                   0x0102
#define WM_MOUSEMOVE              0x0200
#define WM_LBUTTONDOWN            0x0201
#define WM_LBUTTONUP              0x0202
```

```
#define WM_LBUTTONDBLCLK                0x0203
```

Keyboard

Firstly, we defined a `KeyabordEvent` class, which is pretty much like an `Event`, but contains additional information. It has type event (`Press`/`Release` a key), code of button which was interacted with, and character if pressed button is a character key. There is also a class for handling all keyboard events. It contains states of all keys and all types of events.

LISTING 4.6: Class Keyboard (shorten version)

```
class FRTENGINE_API Keyboard
{
public:
    KeyboardEvent onKeyPressedEvent;
    KeyboardEvent onKeyReleasedEvent;
    KeyboardEvent onCharEnteredEvent;
    inline bool IsKeyPressed(unsigned char keycode) const noexcept;
};
```

It is possible to subscribe to those events from somewhere and do anything needed when input occurs:

```
keyboard.onKeyPressedEvent += [this] (Event* event)
{
    // we receive a pointer to base class event
    // which actually refers to KeyboardEvent, so we cast it
    KeyboardEvent* ev = static_cast<KeyboardEvent*>(event);

    if (ev->GetKeyCode() == 'W'))
    {
        // move tetromino down
        // (we will talk more about it later)
        tetromino->MoveY(-2.f);

        // write log to file
        Loger::DebugLogInfo("Tetromino moved down");
    }
}
```

We talk about logging in 4.2.4.

Mouse

As far as for keyboard, we have classes for `Mouse` and `MouseEvent`. The latter contains type which could be one of (`Press`, `Release`, `WheelUp`, `WheelDown`, `Move`, `EnterWindow`, `LeaveWindow`), and `MouseState` (`POINTS` is a struct from `Windows SDK`, it consists of two shorts, for x and y):

LISTING 4.7: Class MouseState

```
enum class FRTENGINE_API MouseButtonType : uint8_t
{
    None, Left, Right, Middle, Other
};
struct FRTENGINE_API MouseState
{
    MouseButtonType buttonType;
    POINTS pointerPosition;
};
```

We will not list here a source of `Mouse`, since it is pretty big. Nevertheless, it contains event instances for all of `MouseEvent` types (just like a `Keyaboard` contains different `KeyboardEvents`). It also stores the last `MouseState`.

4.2.4 Logging

Since we make an application and not a console program, our window is created for rendering a picture for a game, and there is no place for logging there. Moreover, we do not make a game editor program where we could place a window with logs. However, logging is crucial both for development and for the released game. We decided to make logging to a file - fast in implementation, convenient in use. We made three types of log - `Info`, `Warning`, and `Error`. Also there are two sorts of each type - `Debug` and `Release` (or `Deploy`). The former is not written to file if the project configuration is "Release." Even for more gain, we also print the date and time of the log. Example:

```
RLS 2021:04:30 08:26:37 [Info] Create App
RLS 2021:04:30 08:26:37 [Warning] Could not open file 'texture.png',
    using default values
DBG 2021:04:30 08:26:37 [Error] Division by zero
RLS 2021:04:30 08:26:37 [Info] Exiting from app with code 0
```

`DBG` stands for `Debug` which means this message would not be printed in "Release" configuration. And `RLS` means `Release`.

4.2.5 Time

One more vital part of every game engine is a time library. As we saw in 4.2.4, one of its possible usages is to print time in logs to be more aware of when something was going on. We also will use it to drop tetromino every few seconds automatically. Since our needs are pretty simple, so the time library is simple too:

LISTING 4.8: Class Time

```
class FRTENGINE_API Time
{
public:
    static void Init();
    static void Tick();
    inline static float GetSecondsSinceFirstTick();
    inline static float GetDeltaSeconds();
    static LPSYSTEMTIME GetCurrentSystemTime();

private:
    static float _secondsSinceFirstTick, _deltaSeconds;
    static std::chrono::steady_clock::time_point _lastTickTime;
};
```

`Init()` function stores current time `std::chrono::steady_clock::now()` to field `_lastTickTime`. `Tick()` is called after each frame is rendered and stores/calculates values of `_secondsSinceFirstTick` and `_deltaSeconds`. The latter represents how much time has passed since the previous tick. `GetCurrentSystemTime()` returns a special structure which contains current time in convenient format. It has a separate field for every part of date, that is for current year, current day, and so on. The smallest date part is has is current milliseconds. This structure is used in logging (4.2.4).

4.3 Render

4.3.1 Camera

Our Camera does pretty much that what we discussed in 3.2.2. And as we remember, to figure out frustum, and as a result, construct a view matrix, it is mandatory to have $camera position$, $camera look direction$, and $up direction$. So we store those parameters:

```
DirectX::XMFLOAT3 _position, _lookDirection, _upDirection;
```

They will be set on initialization and will be used to form a view matrix. Conveniently, DirectX already has matrix creation implemented:

```
DirectX::XMMATRIX Camera::GetViewMatrix()
{
    return XMMatrixLookToRH(XMLoadFloat3(&_position), XMLoadFloat3(&
        _lookDirection), XMLoadFloat3(&_upDirection));
}
```

Furthermore, we also wrap DirectX function for getting a projection matrix:

```
DirectX::XMMATRIX Camera::GetProjectionMatrix(float fov, float
    aspectRatio, float nPlane, float fPlane)
{
    return XMMatrixPerspectiveFovRH(fov, aspectRatio, nPlane, fPlane);
}
```

Moreover, we provide a possibility to change camera position and rotation in run-time.

4.3.2 Graphics resource

GraphicsResource does not do anything by itself, but is a helpful class. It derives from ITickable (4.1) and stores an address of a Graphics instance. It is also a friend of Graphics so it can provide (delegate) access to some of Ggraphics fields. For instance:

```
inline ID3D12GraphicsCommandList* GetCommandList()
{ return _owner->_commandList.Get(); }
```

4.3.3 Vertex buffer

VertexBuffer is publically derived from a GraphicsResource. Its task is to pass an object's vertices to the Input assembler, just like we described in 3.2.1. Since the vertex structure can be anything, we defined VertexBuffer's constructor as a template:

```
template<typename V>
VertexBuffer(Graphics* owner, V* vertices, UINT verticesNum) :
    GraphicsResource(owner) { // ... }
```

For this and for most of other classes we deleted default constructors, since we do not need them.

VertexBuffer also contains the following fields:

```
ComPtr<ID3D12Resource> _vertexBuffer, _vertexBufferUploadHeap;
D3D12_VERTEX_BUFFER_VIEW _vBufferView;
```

ComPtr<T>is just a smart pointer from Windows SDK. ID3D12Resource is encapsulation of ability of CPU and GPU to write and read from physical memory. It provides an abstraction for working with raw data. D3D12_VERTEX_BUFFER_VIEW is a struct from d3d12.h library:

```
typedef struct D3D12_VERTEX_BUFFER_VIEW {
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes, StrideInBytes;
    }  D3D12_VERTEX_BUFFER_VIEW;
```

D3D12_VERTEX_BUFFER_VIEW is used to work with heaps (ID3D12Resource), D3D12_GPU_VIRTUAL_ADDRES is unsigned 64-bit integer. After setting up heaps and scheduling them for uploading to GPU, we have to save description of all that data to _vBufferView at the end of the constructor:

```
_vBufferView.BufferLocation = _vertexBuffer->GetGPUVirtualAddress();
_vBufferView.StrideInBytes = vertexSize;      // sizeof(V)
_vBufferView.SizeInBytes = vertexBufferSize;  // verticesNum *
    vertexSize
```

Then, when the Render() method of the Graphics will be called, it will call PopulateCommandList() of GameWorld, which will call PopulateCommandList() of every GameObject's instance. Each GameObject's instance will call PopulateCommandList() of each GraphicsResource it has. And then we comes to the VertexBuffer:

```
inline virtual void PopulateCommandList() override { GetCommandList()->
    IASetVertexBuffers(0, 1, &_vBufferView); }
```

We will talk later about what is _commandList. However, we are already pretty aware about IASetVertexBuffer(...), because IA in its name stands for InputAssembler which is known from 3.2.1. So, here we give it vertices that will be used to construct primitives. Let us take a look on this function's signature (it is defined in d3d12.h):

```
void IASetVertexBuffers(
  UINT                             StartSlot,
  UINT                             NumViews,
  const D3D12_VERTEX_BUFFER_VIEW *pViews
);
```

StartSlot is basically an index for device's data where to write buffer, we always write to the first slot (with 0 index). Since we pass only one D3D12_VERTEX_BUFFER_VIEW, we set NumViews to 1.

4.3.4 Index buffer

IndexBuffer is very similar to VertexBuffer except we do need it to be templated:

```
class FRTENGINE_API IndexBuffer : public GraphicsResource
{
public:
    IndexBuffer(Graphics* owner, UINT8* indices, UINT indicesNum);
    inline virtual void PopulateCommandList() override { GetCommandList
        ()->IASetIndexBuffer(&_indexBufferView); }
protected:
    ComPtr<ID3D12Resource> _indexBuffer, _indexBufferUploadHeap;
    D3D12_INDEX_BUFFER_VIEW _indexBufferView;
};
```

Instead of D3D12_VERTEX_BUFFER_VIEW, we have D3D12_INDEX_BUFFER_VIEW:

```
typedef struct D3D12_INDEX_BUFFER_VIEW
    {
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes;
    DXGI_FORMAT Format;
    }  D3D12_INDEX_BUFFER_VIEW;
```

Format is in some way equivalent of StrideInBytes from D3D12_VERTEX_BUFFER_VIEW. It defines which type of data is used and of what size. Since for setting indices we use the array of UINT8 (unsigned 8-bit integer), we set Format to DXGI_FORMAT_R8_UINT. R does not mean anything here, it is just for backward compatibility. 8 stands for size of each element in bits.

4.3.5 Constant buffer

We did not talk about ConstantBuffer in 3.2, as far as it does not influence the rendering directly, and we do not have to pass it to InputAssembler. However, it is still a pretty crucial thing, so we will take a close look at it. ConstnatBuffer is a great example of GPU resource ID3D12Resource. We will process/compute some data on CPU and put in ConstantBuffer which shaders can read from. For example, we will put there *world*, *view* and *projection* matrices, light and material settings. That is everything that is needed to calculate positions of vertices and colors of pixels.

Ideally, it is better to use different buffers for different purposes. For example, use one buffer for light constants since they are the same for all objects on the scene and another buffer for per-object data (rotation, translation, or others). However, for simplicity, we will use only one buffer SceneObjectConstatntBuffer. Every object will own a copy of per-scene constants, which is pretty OK in our case. Moreover, it even helps to optimize data usage. Due to hardware specifics, the data block passed through the constant buffer has to be 256-byte aligned. That is, if our data weights less than 256 bytes, other (256 - *ourDataSize*) bytes will be used for nothing. So, if the total size of the constants is less than 256 bytes, it is better to have one 256-bytes buffer instead of two ones. However, this method has a disadvantage - some constants that are permanent constants through the entire game lifetime will be updated every time we update per-object constants. Nevertheless, that will do for us.

Firstly, we have to define a constant buffer struct that we will work in code with:

LISTING 4.9: Constant buffer structure

```
struct SceneObjectConstantBuffer
{
    DirectX::XMFLOAT4X4 model, viewProj;
    DirectX::XMFLOAT3 cameraPosition;
    FLOAT roughness;
    DirectX::XMFLOAT3 lightPosition1;
    FLOAT falloffStart;
    DirectX::XMFLOAT3 lightPosition2;
    FLOAT falloffEnd;
    DirectX::XMFLOAT3 lightColor;
    float progress;
    DirectX::XMFLOAT4 ambient, diffuseAlbedo;
    DirectX::XMFLOAT3 FresnelR0;
    FLOAT padding[5];
};
```

Besides we align the entire strcut size, we also should align its attributes by 16 bytes. The main reason why do we do this is that C++ packing to heap rules may differ from shader reading form heap rules. That is, the C++ compiler may pack such fields

```
DirectX::XMFLOAT3 lightPosition1, lightPosition2;
FLOAT falloffStart, falloffEnd;
```

in the following way:

```
lightPosition1.x, lightPosition1.y, lightPosition1.z, 0,
lightPosition2.x, lightPosition2.y, lightPosition2.z, 0,
falloffStart, falloffEnd, 0, 0
```

However, if shader will try read read it as following:

```
lightPosition1.x, lightPosition1.y, lightPosition1.z,
lightPosition2.x, lightPosition2.y, lightPosition2.z,
falloffStart, falloffEnd, 0, 0
```

everything will be messed up. Thus, it is a rule of thumb to manually arrange data in constant buffers.

Now, it is time to discuss our ConstantBuffer class. It encapsulates interaction with DirectX constant buffer. Firstly, we create heap, that will be uploaded to GPU. Therefore, we set the property D3D12_HEAP_TYPE_UPLOAD, then resource descriptor is created, it stores the size of the buffer (which is divisible by 256). Then we map the memory of SceneObjectConstantBuffer (typename C) to upload heap. Thus, if we change _buffer, GPU will instantly get those changes. THROW_IF_FAILED is our macros which throws a custom exception when something goes wrong. The exception will be caught in 4.2.1. The next step is to create the constant buffer view in ID3D12DescriptorHeap. So we use cbvSrvHandle to set offset in amount of already created buffers (+ amount of textures, because they both are stored in the same DescriptorHeap) to write the newly created constant buffer view to the correct place. Since the upload heap is mapped to the _buffer, all we need to do to update its data is to update _buffer. In destructor we want to unmap the heap and free memory.

4.3.6 Mesh

Mesh class is considered to store geometry data about an object - its vertices and indices, Index-, Vertex- and ConstantBuffers to upload geometry data to GPU. It also has some additional data, such as its position in the world space or color. This is all data it stores:

LISTING 4.10: Mesh's data

```
std::vector<ConstantBuffer<SceneObjectConstantBuffer>*>
    _constantBuffers;
IndexBuffer* _indexBuffer;
VertexBuffer* _vertexBuffer;
float _radius;
DirectX::XMFLOAT3 _worldPosition;
static const unsigned int _vertexBufferSize, _indexBufferSize;
Vertex _vertices[_vertexBufferSize];
static const std::vector<unsigned char> _indices;
```

Mesh has several methods for initialization and updating buffers. Some of them are: InitializeGraphicsResources(...) and InitializeConstantBuffers(...), UpdateConstantBuffer(...) and PopulateCommandList(). Besides, Mesh is responsible for setting uvs (this is done during specifying vertices positions) and calculating normal for each triangle. To do so, we firtsly have to compute two vectors that lies on triangle's edges: $\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$, $\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$ for the triangle given by its vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$. Then the face normal is calculated as following:

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{||\mathbf{u} \times \mathbf{v}||}$$

It is saved to `Vertex::normal` field. The perfect explanation of calculating `normals` is given in Section 7.7.1 of Lengyel, 2011.

There are two classes that inherits from `Mesh`: `Cube` and `Plane`. They generate geometry data which forms, respectively, cube and plane (actually, a parallelepiped).

4.3.7 Graphics (rendering) pipeline

Class `Graphics` does a lot of `DirectX`-related work which worth a separate paper to be discussed in. We will make a brief overview of its functionality. On initializing phase, it initializes the `Camera`, loads pipeline and assets. In `LoadPipeline()` it describes and creates a `swap chain`, a `command queue`, and create `descriptor heaps`. In `LoadAssets()` `root signature` is created. It also creates the pipeline state which includes loading `shaders`. Here, we specify, how to pass data to the `vertex shader`:

```
D3D12_INPUT_ELEMENT_DESC inputElementDescs[] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
        D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
};
```

`"POSITION"`, `"NORMAL"`, and `"TEXCOORD"` are special semantics for `shader` to know where its data is. We already met `DXGI_FORMAT` in the Section 4.3.4. This one type `DXGI_FORMAT_R32G32B32_FLOAT` means we are dealing with three 32-bit numbers of type float. This is a format of `POSIITON` and `NORMAL`; `TEXCOORD` consists of two numbers.

We define three different `pipeline states` because we have three types of objects.

In `Update` function we wait for previous frame to be rendered. In `Render` we populate command list and execute command queue.

4.4 Graphics

Firstly, we worked with textures. This functionality is still present and is able to work. However, we have discovered a great field for experiments in `shaders`. We already discussed what `vertex shader` (3.2.2) and `pixel shader` (3.2.7) are for. Now, let us take a look at how to implement them. They are written in High-Level Shading Language (HLSL)[2]. We use the latest version `5.0`.

4.4.1 Vertex shader

The first thing we do here is defining the structure of the constant buffer in the same way as in the Listing 4.9. Names do not need to match, though. Here, we use `float4x4` type to represent a matrix of size 4 by 4, and `floatN` for vectors, where `N` is dimension of vector and can have value 2, 3 or 4. For scalars `float` is used.

`register(b0)` points at which register we put this buffer. `Vertex shader`'s `main` function has the following signature:

```
PSInput main(float3 position : POSITION, float3 normal : NORMAL, float2
    uv : TEXCOORD)
```

[2] https://en.wikipedia.org/wiki/High-Level_Shading_Language

PSInput is a custom defined structure. Vertex shader creates, initializes and returns it. Later it will be passed to the pixel shader. Inside the shader we do what we described in 3.2.2. Transform coordinates to world space. We need the 4th element to be 1 because it is a point and we want to save its location during transformation. After transformation 'w' is not needed anymore.

```
PSInput result;
float4 positiomWorld = mul(float4(position, 1), model);
result.positionWorld = positionWorld.xyz;
```

Then transform normal without saving its position relative to world since it is a direction.

```
result.normal = mul(normal, (float3x3)model);
```

Calculate positionnrelative to homogeneous space

```
result.positionHomogeneous = mul(positionWorld, viewProj);
```

Construct custom-defined structures Light and Material and save them to result.

4.4.2 Pixel shader

We have three different pixel shaders for different types of objects: FloorPixelShader.hlsl, BlockPixelShader.hlsl, and BoardWallPixelShader.hlsl. The shader for the floor is almost the same as the one for blocks, except some light constants differ. So we will skip FloorPixelShader.hlsl here.

BlockPixelShader.hlsl

In pixel shader's signature we have to specify special semantic SV_TARGET, so GPU will be able to associate its output with memory where pixel color should be saved.

```
float4 main(PSInput input) : SV_TARGET
```

Since the triangle traversal stage 3.2.5 does not know whether it interpolates normal or texture coordinates, it will not renormalize a normal if the latter becomes a non-unit vector after the interpolation. So, we have to do it by ourselves.

```
input.normal = normalize(input.normal);
```

Then we calculate direction to the camera:

```
float3 toCamera = normalize(input.cameraPosition - nput.positionWorld);
```

Then we calculate lightings (4.4.3) and return a resulting color.

BoardWallPixelShader.hlsl

LISTING 4.11: Pixel shader for board walls

```
float4 main(PSInput input) : SV_TARGET
{
    const float t = InverseLerp(input.progress, input.progress + 0.1,
        input.uv.y);
    const float4 outputColor = lerp(float4(0.38, 0.66, 0.49, 1), float4
        (0.92, 0.32, 0.38, 1), saturate(t));

    // the same code as in BlockPixelShader.hlsl:main(...)

    return litColor;
}
```

`InverseLerp(float a, float b, float v)` is an inverse linear interpolation, that is it returns interpolation *weight* given an interpolated value v on segment $[a, b]$: $w = \frac{v-a}{b-a}$;

`input.progress` indicates our progress on level, range of its values is from 0.0 to 1.0. `input.uv.y` means position of a fragment along y-axis, in range from 0.0 to 1.0. `saturate()` clamps value to range from 0.0 to 1.0. Therefore, `t` will be equal to 0 if position of pixel (in percent, along y-axis) is less than or equal to percent of progress. If fragment's position is in range from `progress` to `progress` + 10%, `t` will be equal to `(pixel position - progress) * 10` and will lie in range 0.0 < `t` < 1.0. And, finally, if fragment y-coordinate is grater than `progress` `t` will have value of 1.0. If we then linearly interpolate between two colors based on value `t` we will get a progress bar with gradient transition with length 10% of object's length. Figure 4.2 shows how do walls look like when progress is 30% - about 30% of wall along its height is green, the rest - is red.

4.4.3 Lighting

Even though the lighting model is very simplified, we will take a quick look at how it works. We will not cover topics that we do not use.

Ambient lighting

In the real world, one of the light sources that illuminate objects is indirect light. That is the light that bounced of some other objects. Until a few years ago, it was almost impossible to compute all those bounds in real-time. Even though now there is hardware capable of doing such a task, it is still costly.

One of the steps to achieving a similar effect to the one we have in the real world is to add `ambient` lighting. It is just a number that we predefine, and it is considered an amount of undirectional light that the object receives. Then an object color is just multiplied by this number: `input.ambient * input.material.DiffuseAlbedo`.

Specular lighting

Some light reflects when it hit the surface, and some refract. Reflected rays are referred to as specular light. Such light reflects in a specific direction, so it may not travel to the eye (camera). Thus, this type of light is dependent on where do we watch it from. Its level of "spreading" depends on the roughness of the surface. The rougher material is, the more scattered the reflected light is.

Point light

In this project, we used only a point light analog to a light bulb from the real world. It illuminates in all directions. This light loses its intensity depending on the distance. A simple way to calculate it is to use a linear falloff function:

$$attenuation = \frac{falloffEnd - distanceToLight}{falloffEnd - falloffStart}$$

Then clamp it to range $[0, 1]$:

$$attenuation = \begin{cases} 1 & attenuation > 1 \\ 0 & attenuation < 0 \\ attenuation & \text{otherwise} \end{cases}$$

On Figure 4.3 is shown how does the flat surface with a low level of glossiness looks like.

4.5 Game world

4.5.1 Game Object

We implemented a simple abstract class to identify every object that can be placed into a scene. It implements interface `ITickable` and declares two new methods: `InitializeGraphicsResources(...)`, `InitializeConstantBuffers(...)`.

4.5.2 Mesh pool

All that buffers creation, uploading, and other things are unjustifiably expensive in terms of CPU/GPU time *if* we create and delete objects every few seconds *and* the total amount of objects is not too big. Whether it is big or not depends on the complexity of objects. Well, puzzle games mostly have a small number of simple objects. Thus, it will be much more efficient to instantiate all those objects on start and hide/move away them (the former is preferable). When we need an object to be on the scene, we just show it.

The solution we chose is to create `MeshPool`, which creates a specified amount of instances of `Cube` (since it operates with pointers to `Cube`'s base class `Mesh`, it can easily be changed to be a template, so it will be able to create a pool of instances of a specified type). It is also responsible for their rendering and destruction.

LISTING 4.12: Most relevant parts of Mesh Pool manager

```
class FRTENGINE_API MeshPool : public GameObject
{
public:
    pair<Result, vector<Mesh*>> GetFreeMeshes(unsigned int amount);
    void ReleaseMesh(Mesh* mesh);
private:
    vector<Mesh*> _meshes;
    vector<bool> _meshUsageFlags;
};
```

Any other game object can "ask" `MeshPool` for some meshes. If there are enough meshes (cubes), game object will receive them, otherwise error code `NotEnoughFreeMeshes` from `enum Mesh::Result` will be returned. When game object does not need a cube anymore, it can give it back to `MeshPool` passing it into a function `ReleaseMesh(Mesh* mesh)`. `MeshPool` will mark in the `_meshUsageFlags` that mesh with a specific index is now free and access to it can be granted to other game objects.

4.5.3 Game world

It is not rational and bug-producing to spawn game objects from anywhere. We add an object which will help us to spawn and destroy `GameObjects`, and will call their `Update()` and `PopulateCommandList()` methods every frame. In Listing 4.13 is shown declaration of some of its functionality.

LISTING 4.13: Game World manages lifetime of Game Objects

```
class FRTENGINE_API GameWorld : public ITickable
{
public:
    template<typename T, class ... Args>
    T* SpawnObject(Args&&... args);
    void DestroyObject(GameObject* object);

    // implements interface ITickable ...
protected:
    std::vector<GameObject*> _gameObjects;
};
```

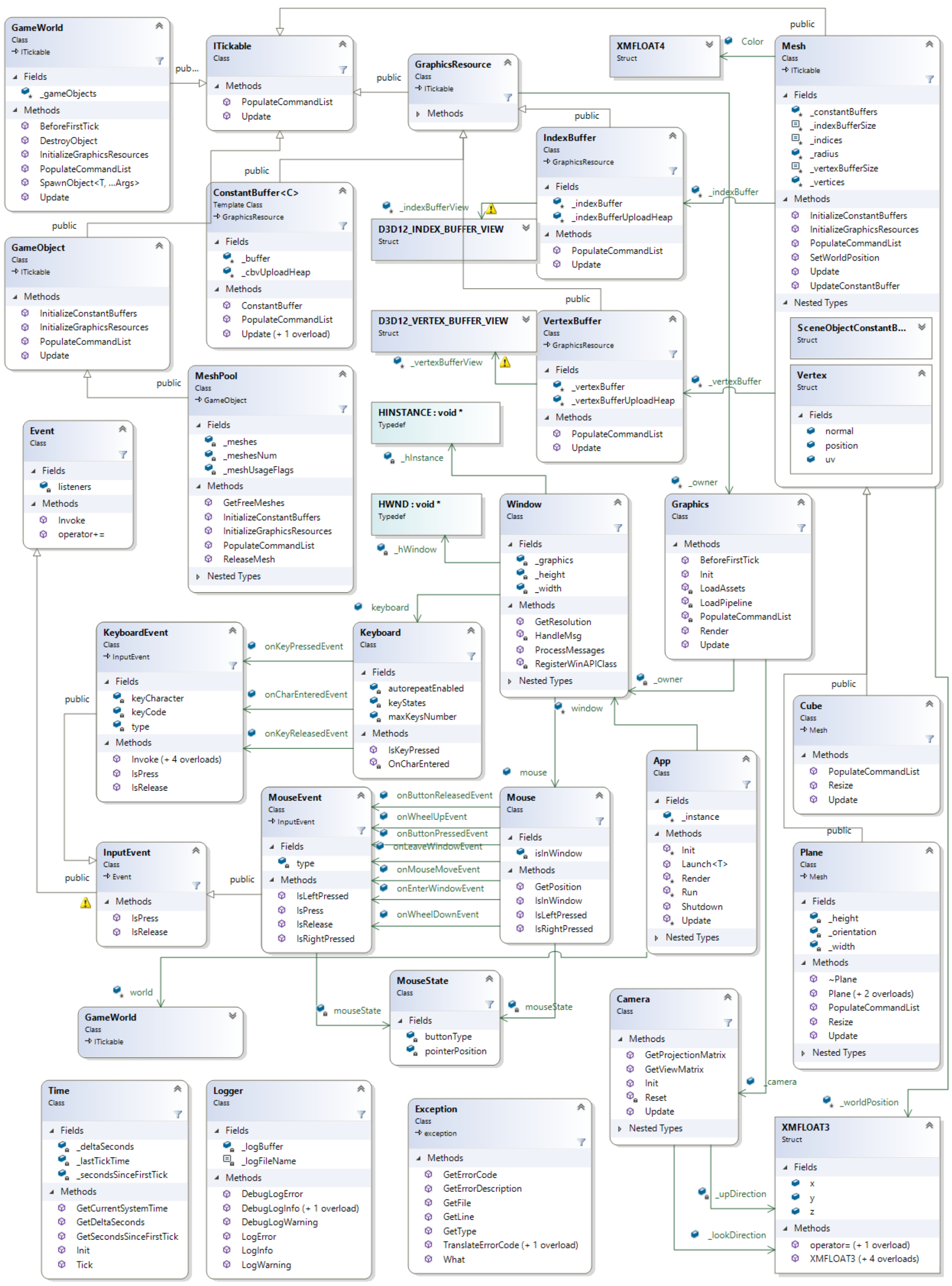


FIGURE 4.1: FRTEngine architecture.

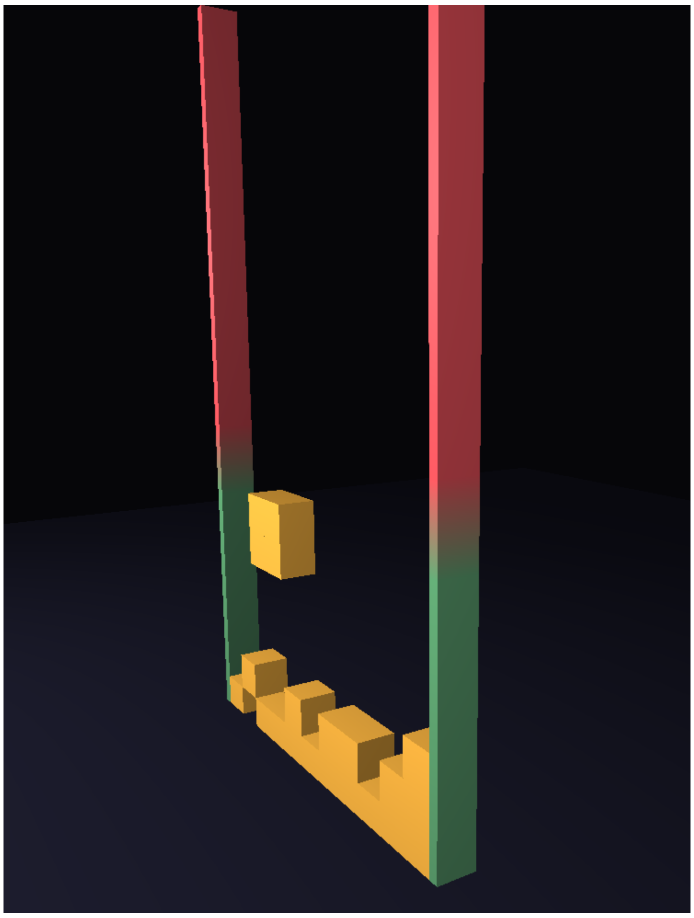


FIGURE 4.2: Board walls are progress bars

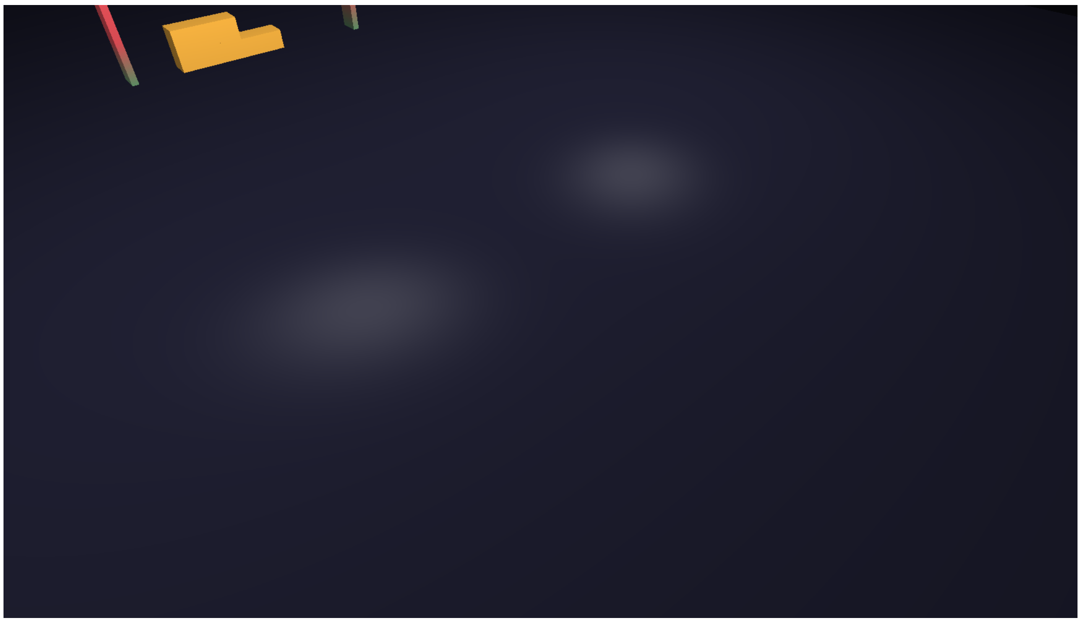


FIGURE 4.3: The surface lit by two point lights

Chapter 5

Usage example - Tetris

5.1 Tetris App

Firstly, we have to create a "main" class. That is `GameApp`, and since we are making a Tetris, it is called `TetrisApp`. In Listing 5.1 are shown some of its functions.

LISTING 5.1: Tetris App

```
class TetrisApp : public App
{
public:
    int Run() override;
    void Update() override;

    // resets a progress of a level and spawns new tetromino
    void Start();

    // resets a progress of a level, clears a board,
    // and deletes a tetromino
    void Reset();
};
```

As we stated in 4.2, it is mandatory for every class, derived from `App`, to implement the `Run()` method. So let us discuss what it does. We have the instance of our `SceneObjectConstantBuffer` as static member of `Tetromino`. Therefore, when the app starts, we set some common values, such as light constants or camera positions. Furthermore, we do not need to do it whenever we want to update an object's constant buffer.

Then, we spawn some objects (`world` is created in `App`'s constructor 4.2):

LISTING 5.2: Spawn game objects in scene

```
world->SpawnObject<BoardBox>();
// amount of cubes to create is 10 columns * 20 rows
_meshPool = world->SpawnObject<MeshPool>(10u * 20u);
 // set board size to has 10 columns and 20 rows with cell size 2
_board = world->SpawnObject<TetrisBoard>(10u, 20u, 2.0f);
tetromino = _board->SpawnTetromino(world, _meshPool, Levels[
    _currentLevel].color);
```

Our next step is to setup necessary handlers for input events. For instance, if we press escape, we exit the app:

```
window->keyboard.onKeyReleasedEvent += [this, &running](Event* event)
{
    KeyboardEvent* ev = static_cast<KeyboardEvent*>(event);
    if (ev->GetKeyCode() == VK_ESCAPE) running = false;
};
```

After all handlers are assigned, some more initializations follows. When everything is ready, we start a game:

LISTING 5.3: Game main loop

```
while (running)
{
    if (const std::optional<int> ecode = Window::ProcessMessages())
        return *ecode;

    Update();
    Render();
}
```

If `if statement` fails, it means something when wrong while processing `Windows (OS)` events. Therefore, the app is terminated.

5.2 Tetromino and Tetris Board

Even though these classes are pretty massive, we will not talk much about them since they are just an implementation of a part of the logic of the original Tetris game [1], and of tetrominoes (Tetriminoes) [2] based on API provided by `FRTEngine`.

5.3 Gameplay

As expected, there are seven types of tetrominoes that are randomly chosen and spawned when needed. They fall by themselves every several moments (depending on a level). A player can rotate a falling tetromino clockwise or counterclockwise, accelerate its falling speed or instantly drop. If one of the board lines is full-filled by blocks (parts of tetromino), it disappears. If such a case occurs, each block from above is moved down precisely by the amount of cleared lines (gravity does not apply). There are ten levels in the game. The color of tetrominoes indicates the change of level. Each subsequent level makes tetromino falling faster. To reach the next level, three lines (a small number is chosen to simplify testing) should be cleared. Progress bar of level progression is built in board walls.

[1] https://en.wikipedia.org/wiki/Tetris
[2] https://tetris.fandom.com/wiki/Tetromino

Chapter 6

FRTEngine vs Unity

Due to its simplicity, Unity is widely used for making games that do not have big and complex worlds and tones of content. Puzzle games fall under the description. Therefore, we decided to reproduce Tetris 3D in Unity. We chose not to completely copying a game logic since it will not make almost any difference. Nevertheless, we completely copied the project's architecture to `C#` - that is, `MeshPool`, `Tetromino`, `TetrisBoard`, `TetrisApp`, and analog for `Mesh`. We also reproduced a scene and walls (Figure 6.1). We added a custom shader to the walls to make the progress bar (however, we turned off the lighting for them). Two point lights are placed on the scene as well. Tetrominoes fall and land and can be moved. That is, the "heavyweight" part is reproduced, as shown in Figure 6.2. We used the default setting for building a project, with the target platform Standalone.

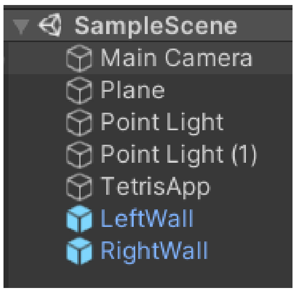

SampleScene
Main Camera
Plane
Point Light
Point Light (1)
TetrisApp
LeftWall
RightWall

FIGURE 6.1: Scene hierarchy in Unity project

We made a performance comparison in a pretty straightforward way, which is accessible for every player. We started both games simultaneously and watched resource usage in *TaskManager*. The numbers were very stable during runtime, and Figure 6.3 shows them. We see that game made with FRTEngine uses a bit more GPU resources than Unity's one. Although it may be a deviation, this probably can be solved by tweaking a rendering pipeline. FRT (FRTEngine) uses as one and half times much as Unity. In fact, it is not a surprise since, for simplicity, we sometimes sacrificed memory usage optimization. Even though not all game mechanics (all game logic including positions calculations and various conditions checking is run on CPU) are implemented on Unity's version, it uses about thirteen times more resources.

Nevertheless, performance is pretty much acceptable and roughly can be considered equal for both games. However, as we mentioned earlier, one of the biggest disadvantages of such game engines as Unity is that they bring into the project a plethora of tools, libraries, or features, that are not actually used. So let us take a look at the size of output files of both programs. We will take into account all necessary files needed for running a game, except system `dlls`. All binaries of the game made with FRT take the place of size 394 KB. While all files needed to run the game

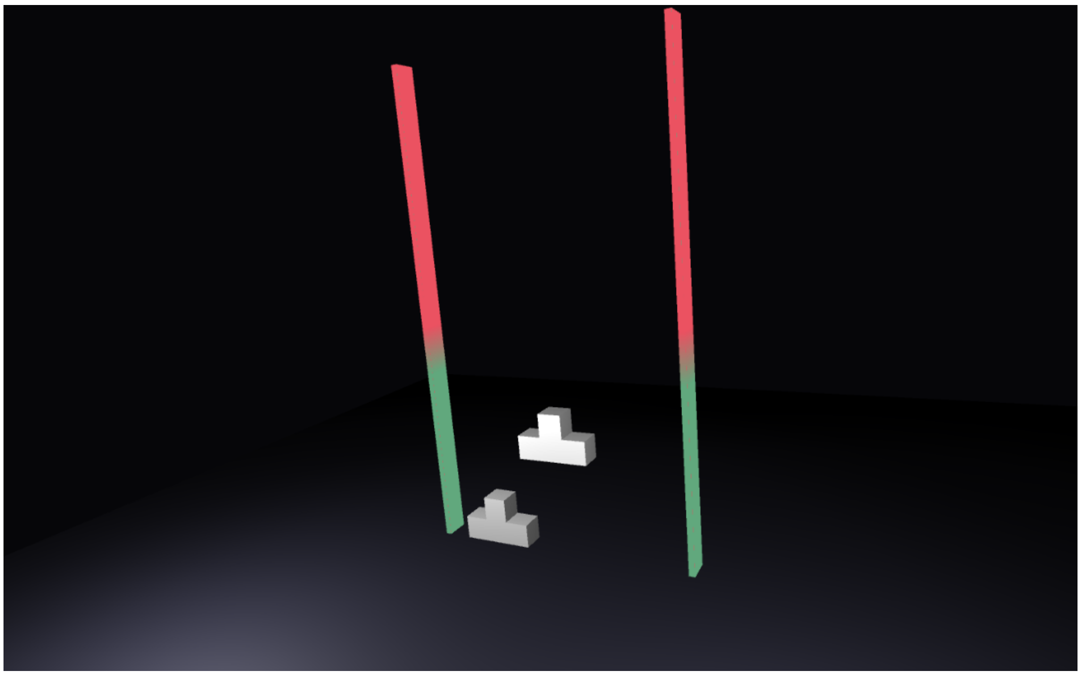


FIGURE 6.2: Tetris 3D made with Unity

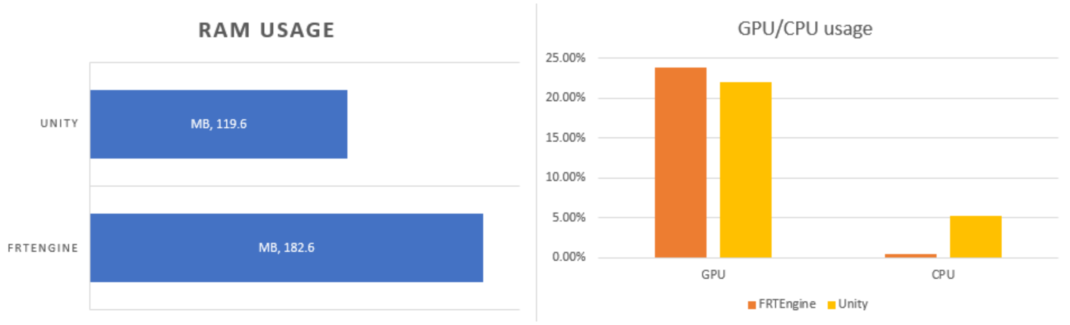


FIGURE 6.3: `FRTEngine` and `Unity` performance comparison
The entry which has a Unity icon represents a game made with Unity

built with Unity (binaries + additional files) are of size 57.5MB, which is almost one and half hundred times bigger than ours.

Chapter 7

Conclusion

7.1 Brief summary

We researched topics about Windows programming and 3D graphics programming. Furthermore, we combined received knowledge and implemented our game engine. We made the Teris-like game with our engine. Then we made a comparison with another game engine, Unity. Even though there are still places for improvements and optimization in FRTEngine, it can already rival Unity in performance when comparing features it is capable of. Moreover, the game made with FRT outweighs Unity in disk space amount required. Consequently, we demonstrated our idea that the specialized solution for *simple* games works better than the universalized one.

7.2 Further improvements

We want to add some new features in the future. For example, loading models from files (such as `.fbx`), shadowing, level editor, convenient multithreading- and math-libraries. To make the engine more flexible, we consider redesigning its architecture.

Bibliography

Gregory, Jason (2018). *Game Engine Architecture*. A K Peters/CRC Press; 3rd edition.

Lengyel, Eric (2011). *Mathematics for 3D Game Programming and Computer Graphics*. Cengage Learning PTR, Third edition.

— (2019). *Rendering*. Foundations of Game Engine Development, Volume 2. Terathon Software LLC.

Luna, Frank (2016). *Introduction to 3D Game Programming with DirectX 12*. Stylus Publishing, LLC.

Microsoft. *Direct3D 11 graphics*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11. (accessed: 05.31.2018).

— *Direct3D 12 graphics*. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics. (accessed: 11.27.2018).