VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Hong Minh Quan Nguyen

DEVELOPING A GAME USING DIRECTX11

Technology and Communication
2022

ACKNOWLEDGEMENTS

I have received much support and advice during the time I spent on this thesis at VAMK, University of Applied Sciences.

I would like to thank Dr. Ghodrat Moghadampour, my supervisor, for his time, patience, and support in helping me complete my thesis.

In addition, I would like to thank all my tutors who have helped me acquire meaningful knowledge during the time I was at VAMK. Furthermore, I would like to thank my family and my friends for their support all the time.

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Hong Minh Quan Nguyen
Title	Developing a Game Using DirectX11
Year	2022
Language	English
Pages	43
Name of Supervisor	Ghodrat Moghadampour

In recent times, game development has become more accessible than ever. Thanks to the groundbreaking breakthroughs and evolutions of commercial game engines, nowadays, anyone with a computer, even a laptop, can build a game in a small amount of time and still achieve satisfactory results. However, they also abstract away how things work under the low level. This thesis aimed to create a sample game without using any game engine to gain a better understanding of how things work underneath.

The game was based on Wolfenstein3D, and it was developed using DirectX graphics API, DirectXTK, and C++. The game contains core parts, such as physics, graphics, and core gameplay functionalities for example various levels, enemies, combat, and pickups.

The final product is a functional game containing two explorable levels. To become a fully complete game, it requires polishing, such as adding sounds, more variety of game modes and objectives, and optimizing the current game systems to fit industry standards.

Keywords	Direct3D, C++, game development, entity framework, computer graphics

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF CODE SNIPPETS

LIST OF ABBREVIATIONS

SDK — Software Development Toolkit

OS — Operating System

HUD — Heads Up Display

GUI — Graphical User Interface

HLSL — High-Level Shading Language

DirectXTK — DirectX Tool Kit

FPS — Frames Per Seconds

GPU — Graphics Processing Unit

VS — Visual Studio

IDE — Integrated Development Environment

MVP — Minimum Viable Product

FoV — Field of View

AABB — Axially Aligned Bounding Box

1 INTRODUCTION

An article from 2021 by WePC estimated that the game industry is worth $178.73 billion, which is an increase of 14.4% from 2020, with recent forecasts estimating the video gaming industry to be worth $268 billion by 2025 /1/. Coupled with the fact that popular game engines such as Unity or Unreal Engine are becoming more accessible with better license deals, user-friendliness, toolboxes, and a great community to help beginners, an increased number of people want to become game developers.

Game engines are very versatile. They contain a variety of tools that help create interactive experiences, for example games, architecture, animation, live events, and automotive. This makes them very versatile and powerful. However, this also adds complexities for hobbyists to get into.

This thesis aims to create a game, using DirectX, a low-level API collection that communicates directly with the hardware. At the end of the process, the programmer should gain a firmer foundation and can confidently start tackling complex game engines, such as Unreal Engine or Unity.

The game is an MVP that mimics Wolfenstein3D, where the player moves around in a maze-like map, shoots enemies, then gains access to the key and progresses to the next level.

This report is separated into the following parts:

- Technologies used
- Application description
- GUI design of the game
- Implementation of the core features of the game
- Testing and production

2 RELEVANT TECHNOLOGIES

This chapter briefly explains the technology and libraries used in this project.

2.1 C++ Programming Language

C++ is a general-purpose programming language, designed with an orientation toward systems programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights. Its key strengths lie in software infrastructure and resource-constrained applications, including desktop applications and video games. /2/

Using C++, developers can apply modern programming paradigms, such as Object-Oriented programming, while still having the power to communicate with low-level hardware like the Graphics driver.

2.2 DirectX

DirectX is a collection of APIs, which provide low-level access to the hardware running on a Windows-based OS. The DirectX SDK consists of runtime libraries in redistributable binary form, accompanying documentation, and headers for coding which aids the development of applications, for example video games, 3D graphics, and simulations on the Microsoft Windows OS and Xbox line of consoles. /3/

2.3 Visual Studio

VS is an IDE that provides multiple tools to enhance every stage of development in C++, including a code editor with IntelliSense and code refactoring, an integrated debugger, a code profiler, a designer for building GUI applications, a web designer, a class designer, and database schema designer. /4/

2.4 DirectXTK

DirectXTK is a collection of helper classes that reduce the amount of boilerplate needed to write DirectX C++ code. /5/

3 APPLICATION DESCRIPTION

This chapter will go through the game flow, how the classes are structured, and the game requirements.

3.1 Requirements Specification

The table below lists all the functionalities planned for the game and their priorities.

Table 1. Feature requirements.

References	Description	Priority (1. Must have, 2. Should have, 3. Nice to have)
F1	Implement map generator	1
F2	Implement component system	1
F3	Implement rendering system	1
F4	Implement collision system	1
F5	Implement hit detection	1
F6	Implement gameplay features (monster, player, pickup)	1
F6	Implement means to transition between different menus, and levels	2

3.2 Use-case Diagram

Figure 1 illustrates how the player can interact with the game world:

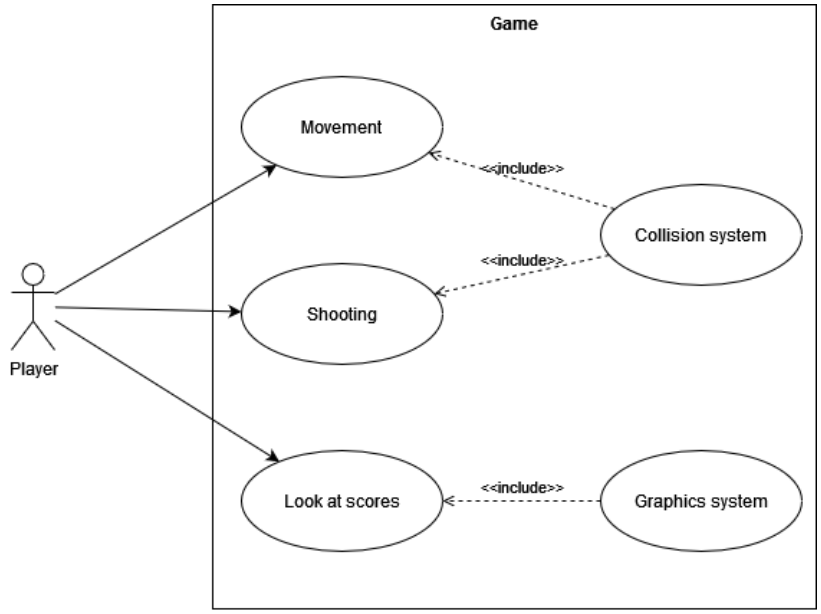

Figure 1. Gameplay use-case diagram.

The actor is the player who wishes to play the game. They can interact with the game world in two ways: moving around with keyboard presses or shooting with mouse clicks. After the player finishes the game, either by completing all the levels or the player's health reaching zero, the game will display the final scores the player has gained.

3.3 Functions Description

Figure 2 below displays in detail the interaction process between the player and the game.

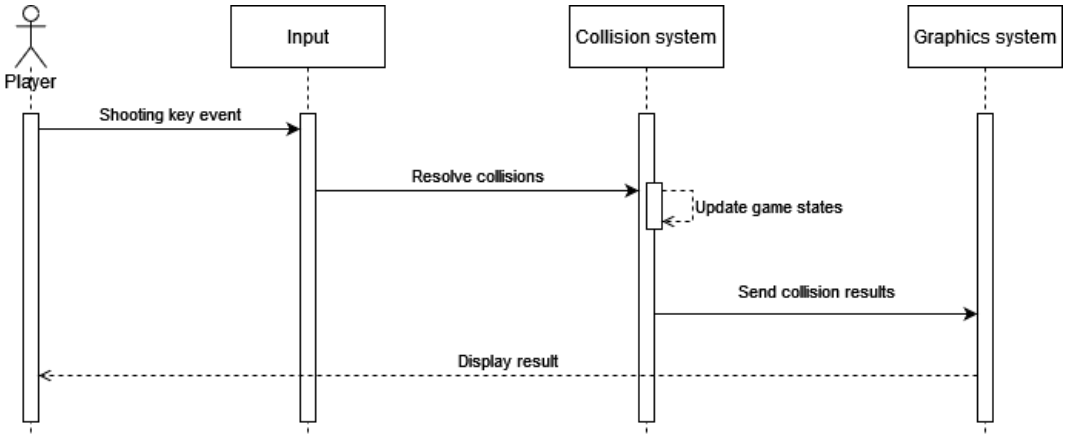

Figure 2. Gameplay sequence diagram.

When the player presses a key or clicks a mouse, the input receives an event. Then, the input processes this event and notifies the collision system. The system resolves any collision that occurred, which in turn updates the game states such as the game scores or the player's health. Finally, the graphic system receives these results and draws the game world that represents the new game status.

3.4 Graphics Pipeline

In general, a graphics pipeline is a series of stages that a GPU goes through to render a 3D model onto a 2D screen. They all boil down to two parts: transform 3D coordinates to 2D coordinates and convert the 2D coordinates into actual-colored pixels. Due to its parallel nature, today's graphics cards have thousands of small processing cores to quickly process data in the graphics pipeline /6/. Graphics APIs such as OpenGL or DirectX abstract the underlying hardware and expose interfaces that programmers can use to configure or modify this process.

3.4.1 Buffers

Every object can be broken down into a list of 3D coordinates called **vertex** data, as seen in Figure 3.

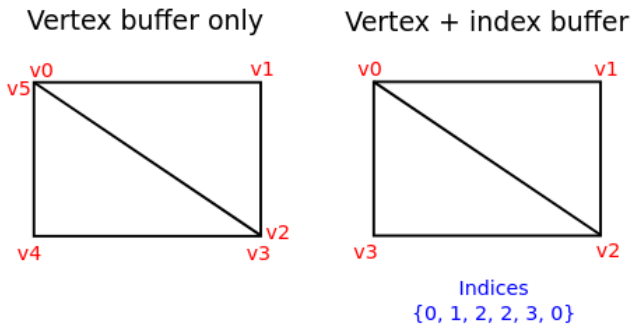

Figure 3. Model represented by vertex and index buffer.

The figure shows that the optimal way to represent a mesh is to use two lists of data, which will be fed to the graphics pipeline /7/. The lists are:

- Vertex buffer: a list of vertices. Each vertex data contains vertex attributes such as position data, color data, texture coordinates, and normal data. /8/
- Index buffer: a list of integer offsets into vertex buffers, so primitives can be rendered more efficiently by reusing existing vertices to form new primitives.

Then, a hint of how the data should be interpreted is provided to the graphics pipeline, for example, line, point, or triangle, usually known as **primitive.** For example, if the hint is a triangle, the vertex data would be formed into a list of three vertices, forming different triangles. Finally, it will be drawn on the display as illustrated in Figure 3.

3.4.2 Shaders

Various stages are programmable to control the style of rendering. These small programs are called shaders, as seen in Figure 4.

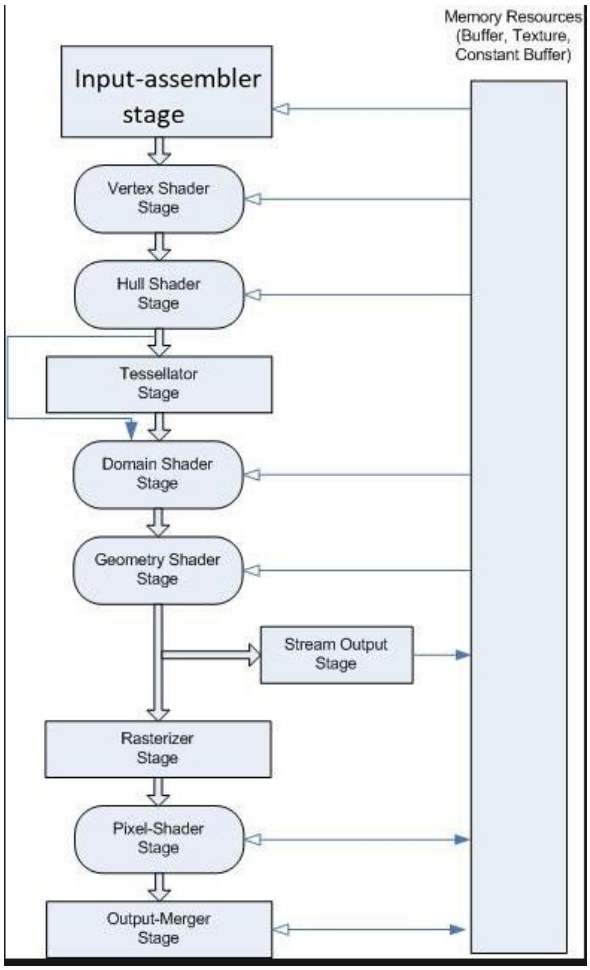

Figure 4. DirectX graphics pipeline.

At each stage, the memory resources are modified, then their outputs are made as inputs for the next stage /9/. At a minimum, the vertex shader and pixel shader must be defined:

- Vertex shader: take each vertex going through the GPU, then process per-vertex operations, such as per-vertex lighting or skinning /10/. Commonly, it is used to transform vertex from 3D to clip space positions through a series of matrices, when combined called the Model-View-Projection matrix. This space makes it easier to clip vertices outside of this range in later stages, which saves processing time /11/.

- Pixel shader: after being rasterized and passing the depth/stencil test, the interpolated per-vertex values, texture data, and other data are used by the pixel shader to compute per-pixel operations, which result in the final color of the screen pixel /12/.

3.4.3 Textures

To give the model a visual look without heavy computational power, a texture is used to "paint" over the model. To specify which part of the texture to sample from, each vertex contains a vertex attribute called texture coordinates /13/.

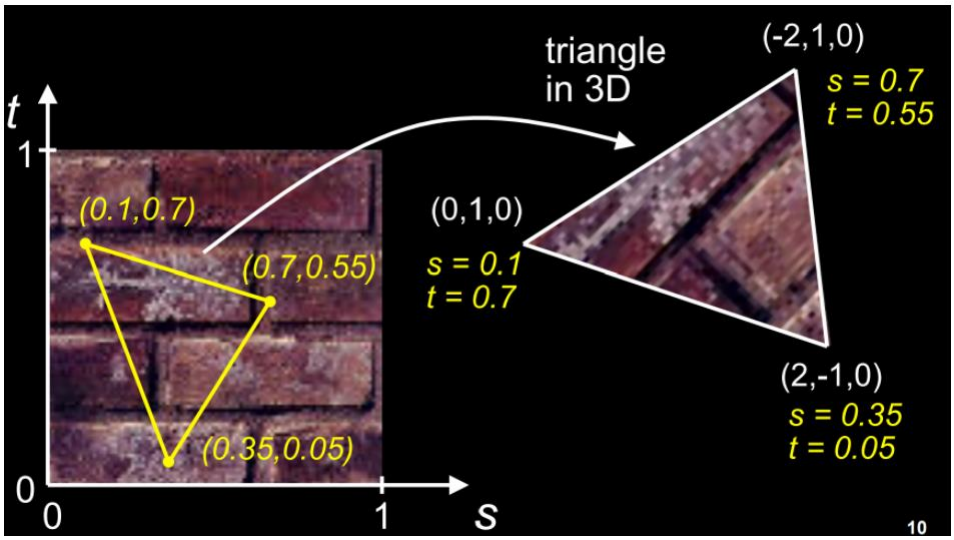

Figure 5. Texture mapping.

Textures are 2D images, and the texture coordinates are denoted $(s,t), 0 \leq s,t \leq 1$. If the values are outside of the range, it would lead to interesting effects, as seen in Figure 6.

Figure 6. Texture behavior.

These wrapping behaviors can be set by the underlying graphics API, and developers have the flexibility to customize them to fit their needs /14/.

3.5 Cartesian Coordinate System

The Cartesian coordinate system is important, since it defines a space that uniquely identifies each point inside it. In this space, a point is specified by numerical coordinates, which are the signed distances to the point from fixed perpendicular oriented lines, measured in the same unit of length. /11/

The concept of the matrix is equally crucial to understand since it is a convenient way to encode information, like translation, rotation, and scale. Each matrix represents a different coordinate system, and they can be combined to conveniently modify coordinates from an initial to a final state. /15/

To display a 3D model, a series of transformations must be performed. The most important ones are the model, view, and projection matrix, detailed in Figure 7.

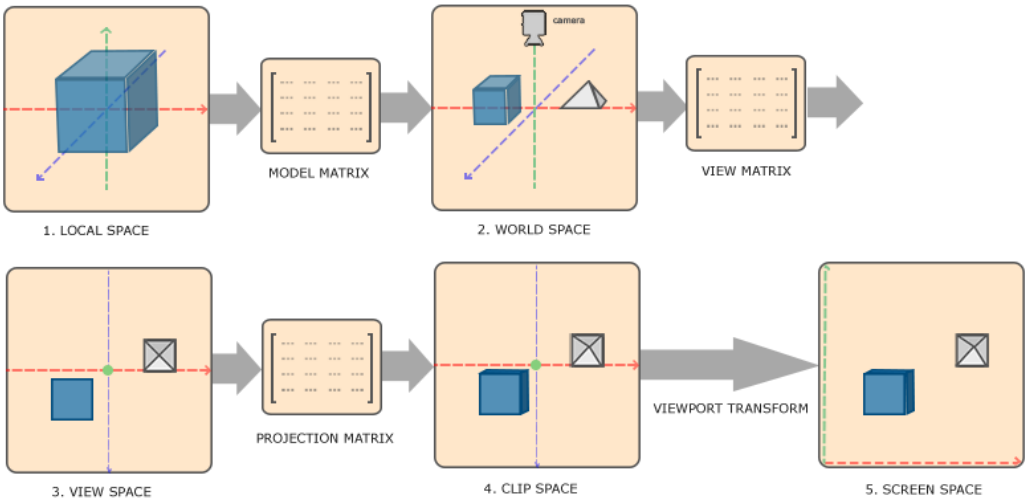

Figure 7. Transformation process of a model.

The reason the coordinates must go through all these spaces is because some operations are easier to perform in a certain space. For example, calculations performed on objects relative to each other make sense in the world space, while modifying the object is easier in its local space. /11/

Each coordinate system is discussed in more detail below.

3.5.1 Local Space

First, a model is created in a 3D modeling program such as Maya3D, or Blender. The vertices are usually relative to the origin. Thus, they are in local space: local to the object. /11/

3.5.2 World Space

After its creation, a model is imported into the application. Usually, the model is positioned somewhere else in a larger world, together with other objects to form a scene. This is a transformation from the local to the world space, which is accomplished with a model matrix.

3.5.3 View Space

The view space is referred to as the eye. It is the space as seen from the camera's point of view. The view matrix is used to transform all objects from the world to the view space, and we can move around a 3D scene by modifying this matrix. /11/

3.5.4 Clip Space

This step marks the end of the vertex shader stage, and the coordinates are expected to be in a specific range, which is the viewing frustum, or the FoV. As the name implies, the graphics pipeline checks and clips anything outside of this range to save power from computing coordinates that are not in the user's view. /11/

After the clipping, the coordinates go through a final operation called perspective division to end up in the NDC space. Only after this stage are the resulting coordinates mapped to 2D screen coordinates and turned into fragments. Fortunately, these steps are all performed automatically by the graphics pipeline. The only thing to consider is which projection matrix to use. /11/

3.5.5 Perspective and Orthographic Projection

There are two types of projection: orthographic and perspective projection. Depending on the use case, each matrix may lead to a different result.

Orthographic projection matrix represents a cube-like frustum, as shown in Figure 8.

This type of projection simply performs clipping and does not affect the coordinates. It is useful for rendering things, such as the player's HUD, or menu since it does not care about depth. /11/

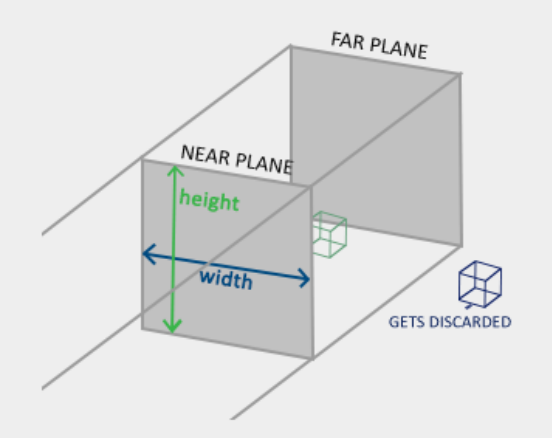

Figure 8. View frustum of orthographic projection.

The second type is perspective projection, which defines a chopped pyramid head shown in Figure 9.

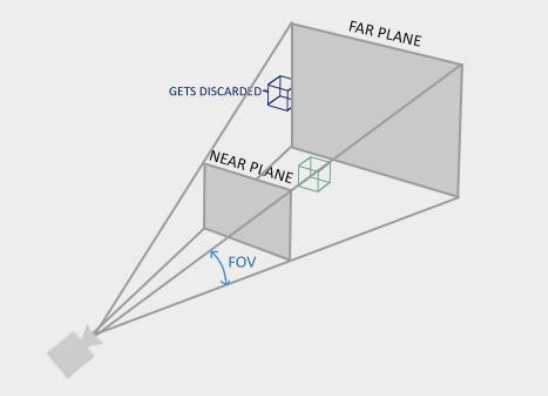

Figure 9. View frustum of perspective projection.

This type of projection is useful for creating realistic 3D scenes since it simulates the concept of perspective: objects that are farther away appear much smaller.

In short, the fundamental steps can be summarized as follows: feed vertex data to graphics pipeline buffers, evaluate a series of transformation matrices and specify shaders. After going through these steps an application can draw and interact with 3D models.

3.6 Component-based Architecture Class Diagram

In game development, it is crucial that the developer can prototype the latest ideas seamlessly. Thus, the composition over inheritance principle is preferred for code reusability and flexibility. A game object is an aggregation of different components, where each component contributes to the behavior of that game object. By using distinct sets of components, we can craft different game objects with vastly different functionalities. The game object also acts as the mediator to communicate between different components.

All game objects will be presented in a level or a scene that can be queried. Going down further, any component knows which game object it belongs to. Thus, it can search for its sibling component by asking its parent game object.

The game object has restrictions to keep it easier for development: each game object only contains one component of each type to avoid repeated functionalities which may lead to unexpected results. Also, components cannot be owned by child components, and game objects cannot be owned by child game objects. This is to make sure that inheritance is one level deep which helps users avoid creating nested and complex structures.

Figure 10 displays an example of this type of component framework.

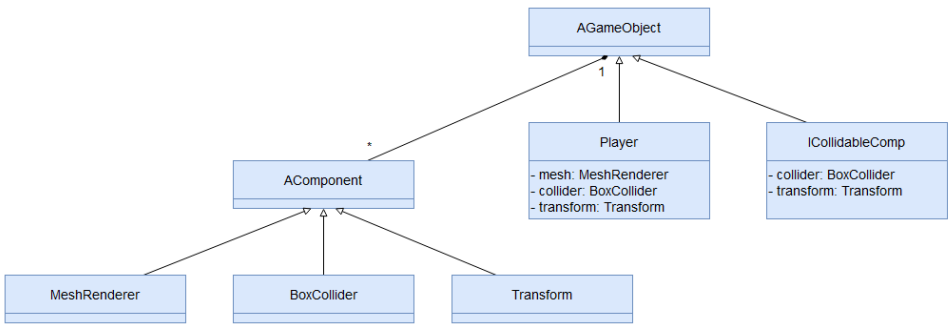

Figure 10. Component-Game Object diagram.

As shown in

Figure 10, **Player** can be placed into the level with **Transform**, can be rendered with **MeshRenderer**, and can process collisions with different game objects that contain **BoxCollider.** One can reuse **Transform** and **BoxCollider** to place a **Trigger-Zone**, which is an empty box that will trigger types of events when **Player** steps into it, for example, a trap or a surprise quest.

3.7 Rendering System Class Diagram

To draw anything on the screen, an application must interact with a graphics API, such as OpenGL or Direct3D. Direct3D is a low-level API that offers rich functionalities to draw primitives on screen, and it specifically targets Windows.

There are multiple Direct3D interfaces required to configure the graphics pipeline. On the other hand, not every class needs to know about these interfaces. Thus, the Direct3D API will be encapsulated by a base rendering class, and any class that implements drawing features will be inherited from this class. Figure 11 shows such an example.

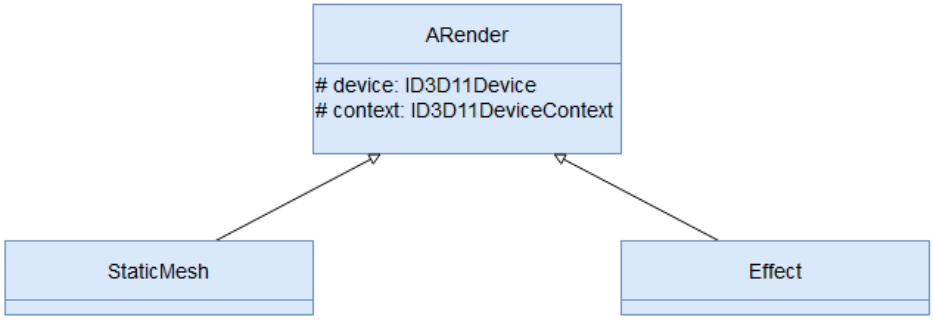

Figure 11. Rendering system class diagram.

Class **ARender** is the base rendering class. It only has one function: to hold references to Direct3D interfaces. Then, the **StaticMesh** class inherits from it to draw a

model on the scene. The **Effect** class implements shader functionality. Therefore, any class that wishes to interact with the graphics pipeline must inherit from **AR-ender**.

3.8 Collision System Class Diagram

A game contains a set of rules, a goal, and participants. With these requirements, the game creates multiple interactions, which makes it interesting and fun. These sets of rules usually come from the physics system.

The collision system ensures that any game object containing a collider component can interact with each other. Also, after a collision between two game objects, there must be ways to notify interested parties, such as the player's HUD drawing a key icon when the player collides with a key game object on a level. The class diagram shown in Figure 12 highlights an architecture that fits such requirement.

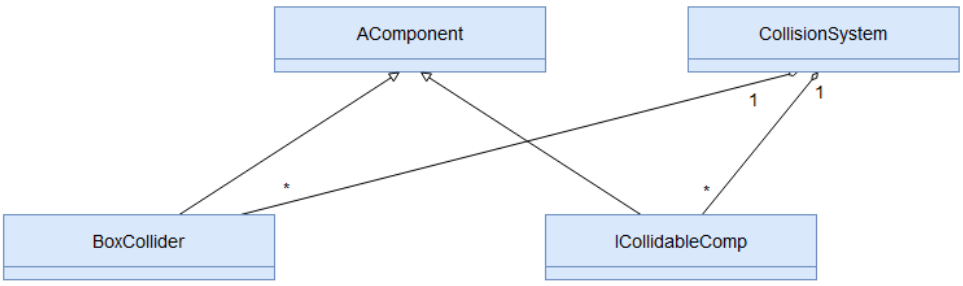

Figure 12. Collision system class diagram.

Class **BoxCollider** is a component that implements collision functionality. This helps prevent two objects from going through each other, like a player going through the wall. **ICollidableComp** class is a simple interface that will be called when a collision event happens. This helps another component in the game object

to perform unique actions that will happen at the appropriate time. The **Collision-System** is responsible for managing the above classes since it will know when these actions should occur.

3.9 Game Flow Diagram

Video games must be interactive, so the game world must live on, even when the user is not present. Thus, there exists a game loop that runs continuously during gameplay. At every loop, it processes the user's input without blocking, updates the game states, then renders it. It tracks the passage of time to control the rate of gameplay. /16/

The game loop process is introduced in Figure 13.

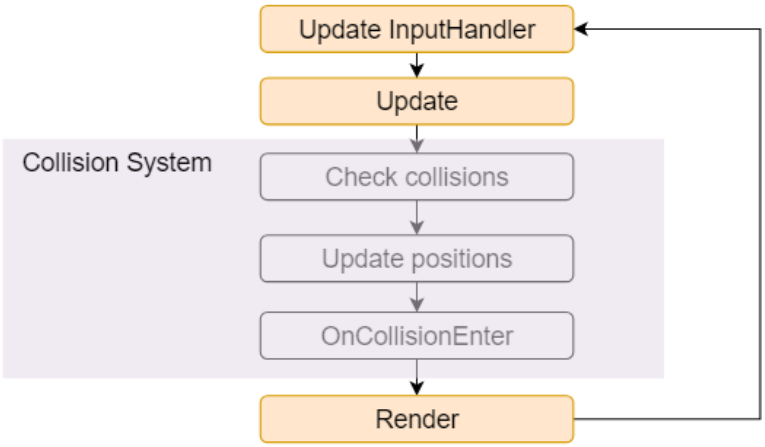

Figure 13. Game Flow diagram.

First, the **InputHandler** processes all user's inputs. Then, all the components will be updated. This may involve firing a weapon, moving around, or triggering an event. Next, the collision system will perform checks between all game objects by updating the positions so that there is no conflict. This means that certain movements performed earlier will be undone.

After these checks, the game should be in a steady state. It is now safe to render the scene. Finally, the game loop is finished and started all over again.

4 ASSETS AND GUI

All assets are stored in the hard disk, with paths relative to the program executable. The asset types used in the game are:

- Textures are stored in the PNG format.
- Fonts are stored in the SPRITEFONT format.
- Shaders are written in HLSL, and they can be precompiled in the compilation stage of the application or loaded at runtime and compiled by the program. In this program, the shaders are precompiled and only read by the program to not slow down when the shaders are loaded.

The game has two simple menus, with a player's HUD.

4.1 Main Menu

The main menu is shown when we start the game. It has a title text to indicate which menu we are currently in, and the instructions to be able to start the game.

Figure 14. Main Menu.

4.2 Game Over Menu

The game over screen is shown, either when the player completes all the levels and finishes the game, or when the player loses, which is when their health drops to zero. The difference lies in the title text, as shown in the following figures.

Figure 15. Game Over Menu when the player finished all the levels.

Figure 16. Game Over Menu when the player lost all health.

4.3 Player's HUD

The player's HUD (Figure 17) presents data that are useful for the gameplay, which contains:

- Current FPS
- The current level the player is at
- Player's current health
- Player's score, which is updated whenever the player kills an enemy
- Whether the player has the key so they can progress to the next level
- A gun model to show where the player's forward direction is

Figure 17. Player's HUD.

When the player shoots, a red muzzle flash appears at the end of the gun's barrel (Figure 18).

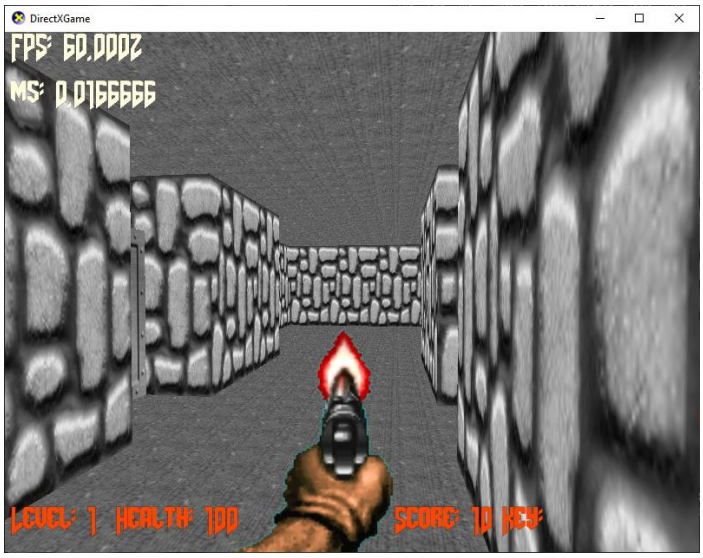

Figure 18. Firing indicator.

The player's health is updated whenever the player is damaged and the screen flashes red, as shown in Figure 19.

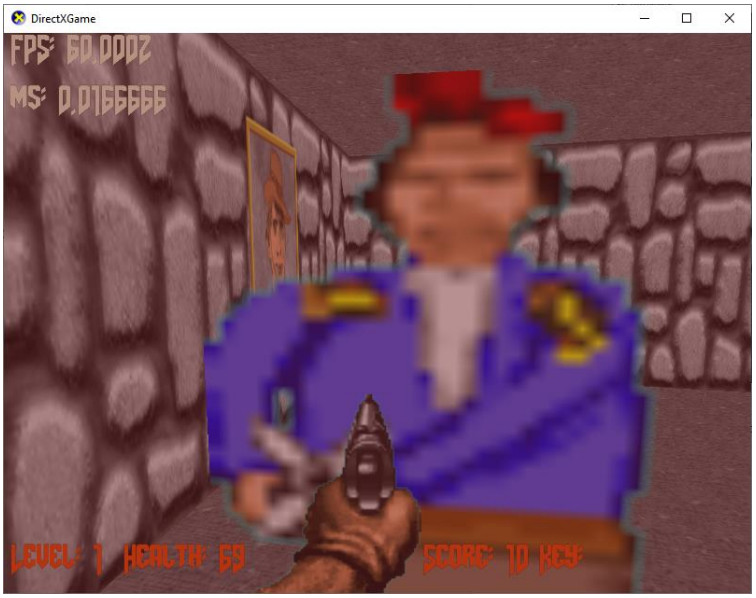

Figure 19. Damage indicator.

5 IMPLEMENTATION

This chapter will explore the implementation details of core features that make up the MVP of the game.

5.1 Setting Up the Environment

The game was developed in VS2019 and tested on Windows 10 computer. To set up the development environment, the following steps were taken:

1. Install VS2019, then tick “Game development with C++” to download the necessary components, which also includes Windows 10 SDK 10.0.19041.0.

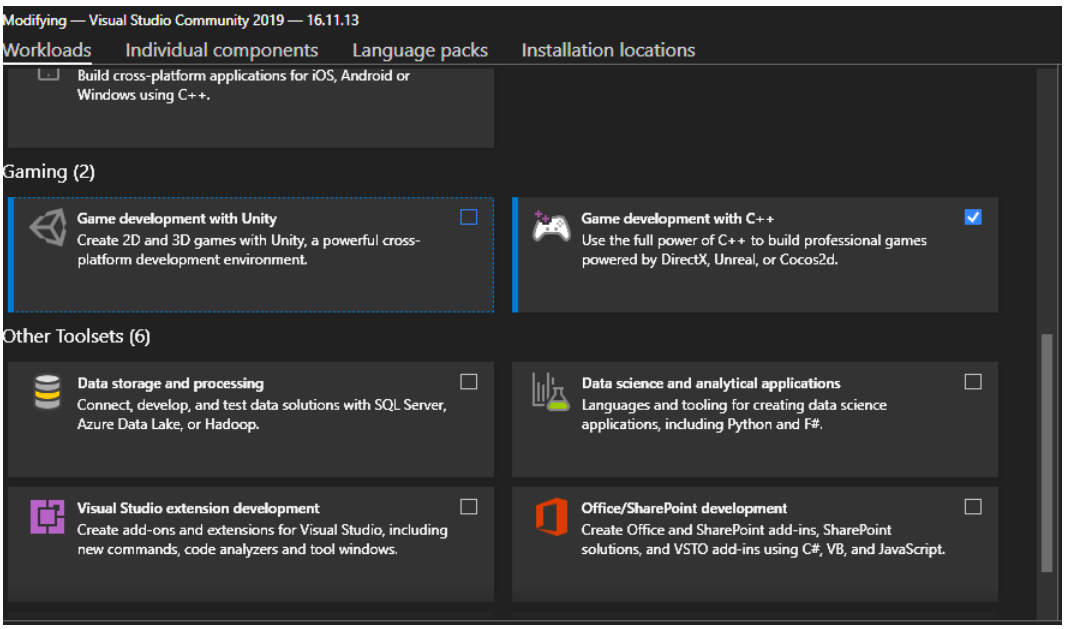

Figure 20. Visual Studio download options.

2. Install Direct3D Game VS project templates, which includes a **Game** class for program initialization, cleanup, and the core game loop. It also provides access to helper classes to gain a jump-start in developing games, e.g., a **StepTimer**, and **DeviceResources** to access DirectX components. /5/

3. Follow the steps in “Adding the DirectXTK” tutorial to integrate DirectXTK classes to the project. /17/

5.2 Overview of the Structure

The core features that make up the game architecture are:

- **Rendering:** classes that interact with Direct3D system
- **Collision:** components managed by the collision system
- **GameObject** and **Component:** classes that make up the game functionalities

Every functionality will be bootstrapped by these lower layer pieces.

5.3 Component System

As described in the section earlier, component-based architecture is preferred due to flexibility and reusability in game development. This section will go through the core functionalities to create this architecture.

5.3.1 AComponent Class

The basic building block of a component-based architecture is the **AComponent** class. It is a simple class with an empty implementation so that its derived classes can add functionality to it. The following code snippet highlights the class’s header file.

```
class AComponent
{
public:
    GameObject* m_owner = nullptr;
    AComponent() = default;
```

```
    virtual ~AComponent() = default;
    virtual void Init() {}
    virtual void Update(float) {}
    virtual void Render() {}
};
```

Code Snippet 1. AComponent header file.

Each **AComponent**-derived class aims to provide only one functionality. This promotes modularity, which has some benefits: easy to develop, easy for the programmer's cognitive load, reusable components in various places, to name a few.

A challenge that arises is that the operations of one component need to be combined with the operations from another component, for all of them to work together and create complex interactions. In short, components need to know how to communicate with each other. This concern is addressed by the **GameObject.**

5.3.2 GameObject Class

GameObject is a generic container that contains and manages its **AComponent.** Each **AComponent** contributes a function, and together they make up the in-game logic of that **GameObject.** The code snippet below shows the basic functionalities of the class.

```
void GameObject::Init()
{
    for (auto& c : m_components)
        c->Init();
}

void GameObject::Update(float dt)
{
```

```
    for (auto& c : m_components)
        c->Update(dt);
}

void GameObject::Render()
{
    for (auto& c : m_components)
        c->Render();
}
```

Code Snippet 2. GameObject parent class implementation.

The class has the responsibility to loop through all the components it has and execute the necessary function on each of them.

The second responsibility that **GameObject** class does is how it manages the components, as shown in Code Snippet 3.

```
template<typename T>
T* AddComponent(std::unique_ptr<T> component)
{
    assert(!Contain<T>());
    T* ret = component.get();
    component->m_owner = this;
    m_components.push_back(std::move(component));
    // Add to bitset and array for later queries
    m_componentArr[GetComponentTypeID<T>()] = ret;
    m_componentBitset[GetComponentTypeID<T>()] = true;
    return ret;
}
```

```
        template<typename T>
        T* GetComponent() const
        {
            assert(Contain<T>());
            AComponent* ret =
m_componentArr[GetComponentTypeID<T>()];
            return reinterpret_cast<T*>(ret);
        }
```

Code Snippet 3. Querying functions of **GameObject.**

As indicated in Code Snippet 1, each **AComponent** has a reference to its owner, the **GameObject**. So, it can synchronize with its sibling components by asking the **GameObject** to give a reference to the component it wants.

When a component is added to the **GameObject**, it is assigned a type identifier. This identifier can be used to look up the correct component in an array. There will not be any collision, since each **GameObject** only has one **AComponent** of each type.

5.4 Rendering System

The purpose of the rendering system is to help decouple the complexity of rendering things on screen, by providing classes that hide the implementation details so that the component system can easily integrate and use them.

5.4.1 StaticMesh Class

Before talking about the **StaticMesh** class, the **ARender** class deserves a mention.

```
    class ARender
    {
```

```
    public:

        static void Init(ID3D11Device1* device,
ID3D11DeviceContext1* context);

    protected:

        static ID3D11Device1* s_device;

        static ID3D11DeviceContext1* s_context;

    };
```

Code Snippet 4. ARender parent class header file.

This class contains references to two important objects:

- **Device**: a factory that allocates resources to be used by the GPU. Usually, the process is to specify data on the application side, then use the device to allocate resources and assign the data to those resources. /18/
- **DeviceContext:** provides means to set pipeline state and generate rendering commands using the resources owned by the device. /18/

So, a class such as **StaticMesh** can inherit from the **ARender** class to use the above objects to draw things on a Windows OS machine , as shown in the snippets below.

```
    void StaticMesh::CreateD3DBuffers()

    {

        DX::ThrowIfFailed(CreateStaticBuffer(s_device,
m_vertices,3D11_BIND_VERTEX_BUFFER, &m_hwVertices));

        DX::ThrowIfFailed(CreateStaticBuffer(s_device, m_indices,
D3D11_BIND_INDEX_BUFFER, &m_hwIndices));

    }
```

Code Snippet 5. Create3DBuffers() method

```
    void StaticMesh::Render()

    {

        UINT stride = sizeof(VertexPositionTexture);
```

```
        UINT offset = 0;

        s_context->IASetVertexBuffers(0, 1,
m_hwVertices.GetAddressOf(), &stride, &offset);

        s_context->IASetIndexBuffer(m_hwIndices.Get(),
DXGI_FORMAT_R16_UINT, 0);

        s_context->IASetPrimitiveTopology(

            D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

        s_context->DrawIndexed((UINT)m_indices.size(), 0, 0);

    }
```

Code Snippet 6. Render() method implementation of **StaticMesh** class.

A static mesh contains a set of vertices that can be cached into video memory, to be rendered by the graphics card. First, the vertices are created in video memory thanks to **Device** and its API function, **CreateStaticBuffer()**. Then, when it is time to render the object, these allocated buffers are referenced to **DeviceContext** by binding to it. With the data, the **DeviceContext** can issue a draw call to draw this object on the screen, through the API function **DrawIndexed()**.

5.4.2 MeshRenderer Class

This class is the glue between the rendering classes that interact with Direct3D interface and the component system. It inherits from **AComponent**, and overrides the **Render()** method, as shown below:

```
    class MeshRenderer : public AComponent

    {

    public:

        MeshRenderer() = default;

        MeshRenderer(

            std::unique_ptr<Rendering::StaticMesh> mesh,

            std::shared_ptr<Rendering::UnlitEffect> effect);
```

```
        ~MeshRenderer();


        void Render() override;
```

Code Snippet 7. MeshRenderer class header file.

The class holds a reference to a static mesh object and an effect. This effect is a shader's interface provided by DirectXTK. Then, in the Render() method's implementation, these objects are put into use:

```
    void MeshRenderer::Render()

    {

        if (m_owner)

            m_effect->SetWorld(m_owner->m_transform-
>GetGlobalTransform());

        else

            m_effect-
>SetWorld(DirectX::SimpleMath::Matrix::Identity);



        m_effect->SetProjection(

            Game::Instance()->GetLevel()->GetMainCamera()-
>GetViewProjection());

        m_effect->Apply();

        m_mesh->Render();

    }
```

Code Snippet 8. Render() method implementation of **MeshRenderer** class.

Since all the challenging work is abstracted away, the **MeshRenderer** job is quite simple. It simply sets the correct transformations for the shaders and calls the rendering commands to render the mesh on the screen.

5.5 Collision System

To create meaningful interactions between different game objects, a collision system needs to be put in place. There are three classes to note: **ICollidableComp**, **BoxCollider**, and **CollisionSystem**. Each class plays a role, and together they make up the collision system.

Furthermore, every collider object will be on a layer, and it can specify which layer is allowed to collide with it, as demonstrated in Code Snippet 9.

```
enum class CollisionLayer : uint32_t
{
    kDefault = 1,
    kPlayer = 2,
    kDoor = 4,
    kMonster = 8,
    kConsumable = 16,
};
```

Code Snippet 9. CollisionLayer Enum.

By adding this restriction, classes can be customized easily, for example, removing friendly-fire damage between friendly enemies.

5.5.1 CollisionSystem Class

The **CollisionSystem** class is responsible for orchestrating and managing all objects that contain **ICollidableComp** component. Its **Update()** method is invoked at the end of the update step when all objects in the level have been updated. Code Snippet 10 shows the implementation of the method.

```
void CollisionSystem::Update()
{
```

```
        for (auto& c : s_colliders)
        {
            if (!c->m_isStatic)
                c->UpdateTheoreticalPosition();
        }
        Resolve();
    }
```

Code Snippet 10. Update() method of **CollisionSystem** class.

The code checks if the collider is static, for example, a tree or a wall. If it is not, then it will update the object's position in this frame. If the positions have been updated, it will call the **Resolve()** method, whose implementation is detailed below in Code Snippet 11.

```
    void CollisionSystem::Resolve()
    {
        for (int i = 0; i < s_colliders.size(); ++i)
        {
            BoxCollider& box1 = *s_colliders[i];
            if (box1.m_isStatic)
                continue;

            for (int j = 0; j < s_colliders.size(); ++j)
            {
                BoxCollider& box2 = *s_colliders[j];
                if (box1.m_owner->GetID() == box2.m_owner-
>GetID())
                    continue;
```

```
                bool isLayerCollide =
                    (bool)(box1.m_collidingLayer & box2.m_layer);
                if (!isLayerCollide)
                    continue;

                CollisionData cd = box1.Intersects(box2);
                if (cd.isColliding)
                {
                    box1.ResolveOverlap(cd);
                    box1.m_owner->OnCollisionEnter(&box2);
                    box2.m_owner->OnCollisionEnter(&box1);
                }
            }
        }
    }
```

Code Snippet 11. Resolve() method of **CollisionSystem** class.

Two loops are taking place to check the collision between every two different objects. Any static object, a comparison is made between the same object, or if two objects are not colliding, then that loop is simply skipped. After making sure that two objects are colliding with each other, then it resolves the overlap, so they do not collide anymore, and calls the event **OnCollisionEnter()** on the two collided objects.

5.5.2 BoxCollider Class

BoxCollider is responsible for resolving the actual collision, which lies in the **ResolveOverlap()** method shown in Code Snippet 12.

```
void BoxCollider::ResolveOverlap(const CollisionData& cd)
```

```
    {
        assert("Static objects should have been skipped" &&
!m_isStatic);
        Vector3 move = m_owner->m_transform->GetPos() - m_oldPos;
        m_owner->m_transform->Translate(move * cd.collision);
        m_x = m_owner->m_transform->GetPos().x;
        m_y = m_owner->m_transform->GetPos().z;
        if (m_type == ColliderPivotType::kCenter)
        {
            m_x -= m_w * 0.5f;
            m_y -= m_h * 0.5f;
        }
    }
```

Code Snippet 12. ResolveOverlap() method of **BoxCollider** class.

The class keeps track of the object's position before and after the update step of the game loop. If there was any overlap that occurred, the position is simply reverted.

5.5.3 ICollidableComp Class

The **ICollidableComp** class is also an important piece, despite its simplicity. The header file implementation is shown in Code Snippet 13.

```
    class ICollidableComp : public AComponent
    {
    public:
        virtual void OnCollisionEnter(BoxCollider* other) = 0;
```

```
};
```

Code Snippet 13. ICollidableComp header file.

The class was inherited from **AComponent**, so that it can interact with the **GameObject** that owns it. Therefore, the derived classes of **ICollidableComp** can reference other components of the **GameObject** to implement the **OnCollisionEnter()** event, such as recover health when the player picks up a health kit, or the key is obtained which allows the player to reach the next level.

5.6 Map Generator

The map was a 3D map generated from a 2D image. This makes it easy to customize the map without having to create a fully complex map editor such as those found in game engines.

First, a format had to be decided for the map. Table 2 below shows the color code of each object.

Table 2. Color code of each object.

Color	Object
Black	Wall
Red	Wall texture
Green	Ceiling and Floor texture
Blue	Item's spawn location

The next step was to define a way to load the texture in accordance with the user's choice. In this case, a sprite sheet consisting of 16 textures was used to define the look of the map.

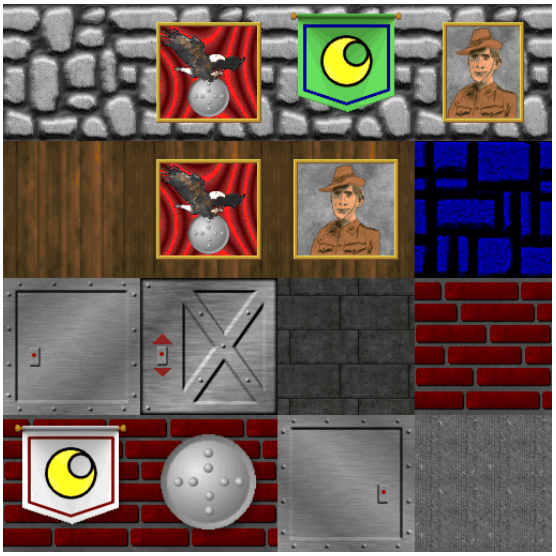

Figure 21. Map textures.

The final requirement is the map image itself.

Figure **22** shows an example of such an image.

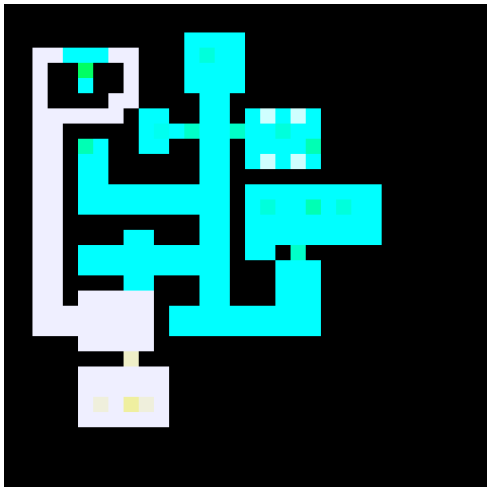

Figure 22. Example map image.

The image describes all the spawn locations of each item, the map layout, and where to place what texture.

Having gathered all the specifications, the next step was to provide a method that can map the color value to the desired texture. Code Snippet 14 shows the implementation of this algorithm.

```
    void Level::ComputeTexCoords(
        uint32_t color, uint32_t channelFlag,
        uint32_t bitsToShift,
        float* uLower, float* uHigher, float* vLower, float*
vHigher)
    {
        int texLoc = ((color & channelFlag) >> bitsToShift) / 16;
        int u = texLoc / 4;
        int v = texLoc % 4;
        float xSize = 0.25f;
        float ySize = 0.25f;
        float xOffset = u * xSize;
        float yOffset = v * ySize;
        *uLower = xOffset;
        *uHigher = xOffset + xSize;
        *vLower = yOffset;
        *vHigher = yOffset + ySize;
    }
```

Code Snippet 14. Algorithm to choose texture for the wall/ceiling/floor.

The code first extracts the correct color channel from the integer value residing in the pixel of the map image. Then, it is mapped from range $[0, 255]$ to range $[0, 15]$ with a simple division. Next, the values are mapped from range $[0, 15]$ to range

$[0, 3]$, since there are four rows and four columns. Finally, the values are mapped to range $[0, 1]$ to match the texture coordinates that will be passed to the graphics pipeline.

Figure 23 shows an example of the algorithm in action.

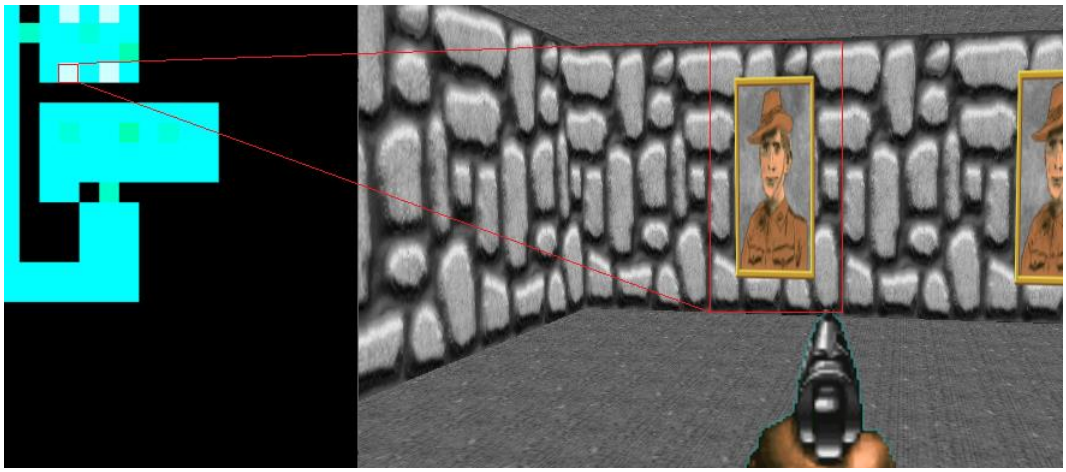

Figure 23. An example of the texture choosing algorithm.

The color pixel of the map image has the value of $(207, 255, 255)$ in format (R, G, B). With $R = 207$, the wall texture used corresponds to the upper-right texture in Figure 21. Likewise, $G = 255$ means that the ceiling and floor use the texture lies in the bottom-right location of Figure 21.

6 TESTING GAMEPLAY FEATURES

Testing plays a vital role in any software development cycle. Test documentations help keep track of the critical bugs that need to be addressed and verify how the game works. Test cases are created to keep track of the set of functions to be tested described in the following table.

Table 3. Test cases.

ID	Test cases	Testing steps	Expected
1	Correct textures are generated	1. Open map image	Image is opened
		2. Change red and green channels	Pixel is changed
		3. Execute the game	Wall, ceiling and floor are different
2	Correct item's spawn locations	1. Open map image	Image is opened
		2. Change blue channel	Pixel is changed
		3. Execute the game	Item's spawn location has changed
3	Player collision	1. Move towards wall	Player collides with wall
		2. Move towards monster	Player collides with monster
4	Monster collision	1. Let monster moves towards wall	Monster collides with wall
5	Door implementation	1. Player/Monster moves towards door	Player/Monster collides with the door
		2. Door is opened	Player/Monster moves to next room
6	Monster gameplay features	1. Monster sees player	Monster follows player
		2. Monster shoots player	Player's health is dropped
		3. Monster's health reaches 0	Monster plays death animation and vanishes
7	Player gameplay features	1. Player shoots monster	Monster's health is dropped
		2. Player collides with pickups	Pickups disappear, and player gains effects
8	Health Pickup features	1. Player collides with Pickup	Player restores health, if it is below 100
9	Key Pickup features	1. Player collides with Pickup	Player can progress to next level
10	Exit Door implementation	1. Player collides with door	Player goes to next level
11	Game Over scene	1. Player's health reaches 0	Game Over scene is shown
		2. Player finishes all levels	Game Over scene is shown
11	Main Menu scene	1. Press enter button	First level is launched

These tests were performed on a Windows 10 machine. Each test case covered a different functionality of the game. After thorough testing of each feature, the game was play-tested multiple times to make sure that the game can be completed without any game-breaking bug occurring.

6.1 Play Test Process

First, the player is greeted with a Main Menu scene.

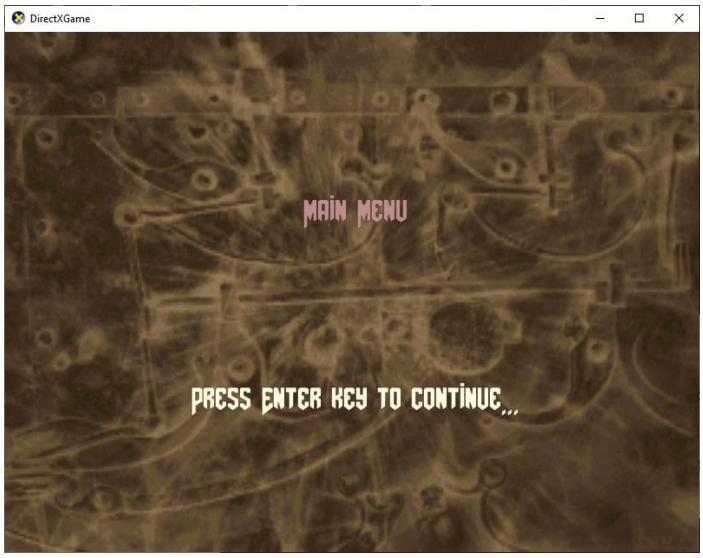

Figure 24. Main Menu.

When the player presses the “enter” key, the game enters the first level.

Figure 25. First level.

As the player progresses through the level, the player will encounter enemies. Figure 26 shows such an example.

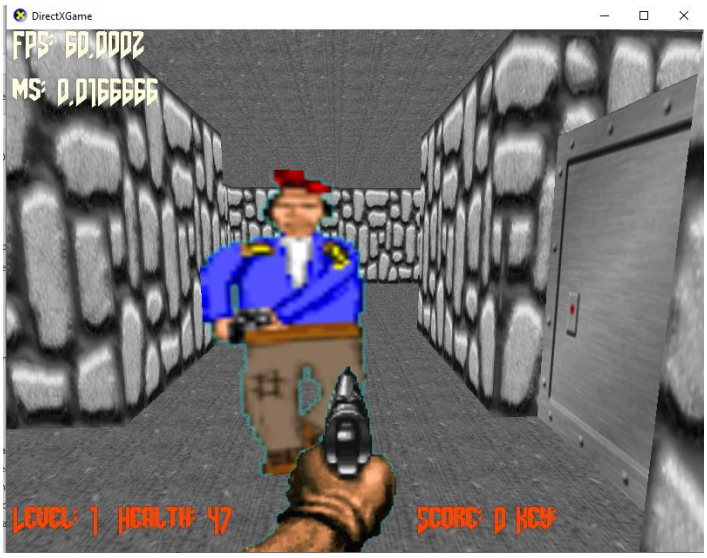

Figure 26. Encounter an enemy.

An enemy can damage the player, which is indicated by two things: the scene splashes red, and the HP HUD changes its value, see Figure 27.

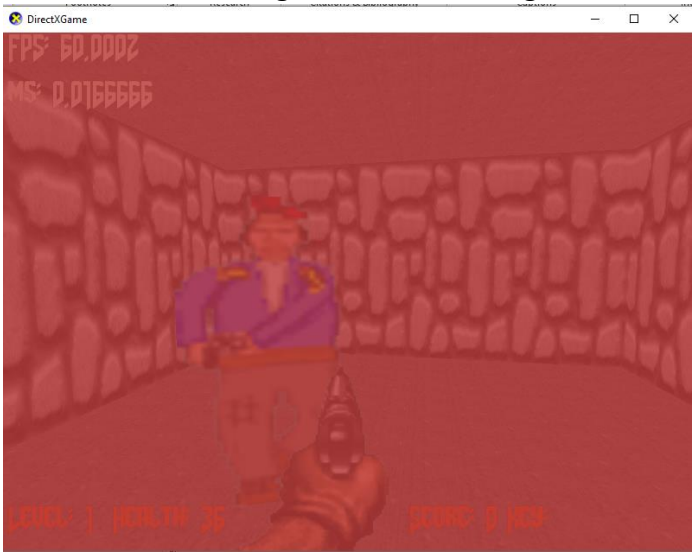

Figure 27. Player takes damage.

Each room the player enters may contain a pickup such as a health pack. There will always be a room that contains a key that the player can pick up, as shown in Figure 28.

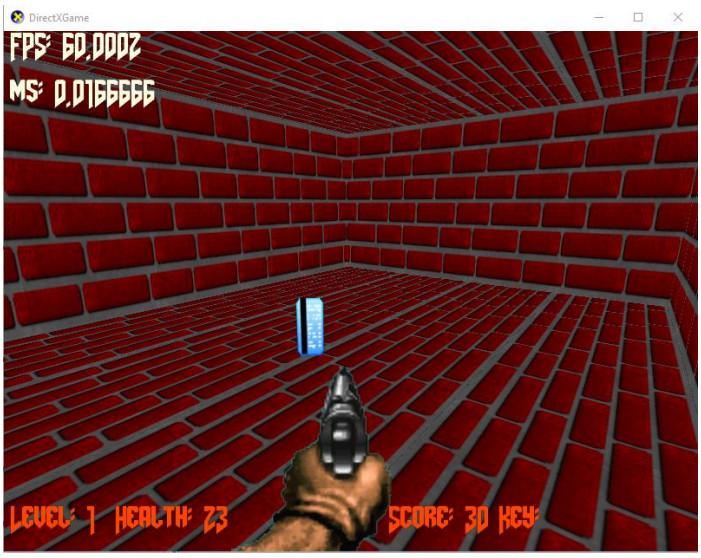

Figure 28. Encounter key.

Finally, the player can find the exit door, and if the player has the key, as detailed by the key icon on the HUD, the player can progress to the next level (Figure 29).

Figure 29. Exit door.

If the player's health reaches zero during the gameplay, a Game Over scene will appear (Figure 30).

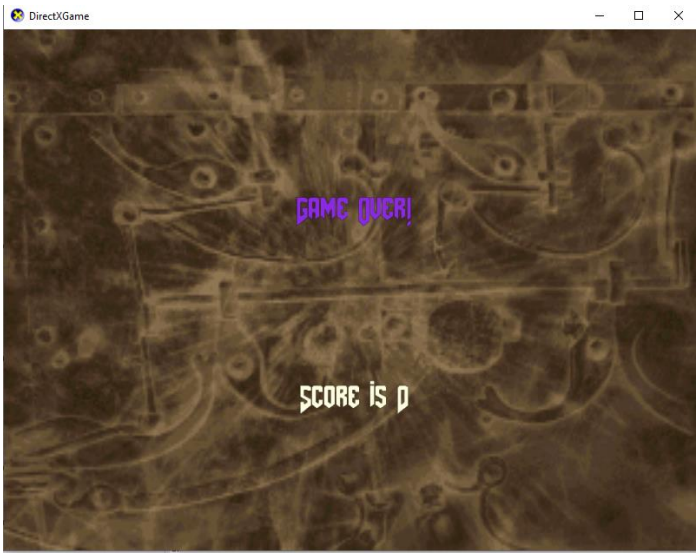

Figure 30. Game Over scene.

On the other hand, if the player finishes all levels, a Game Over scene with different texts will appear, see Figure 31.

Figure 31. Game Over scene when the player finished all levels.

The player can choose to start all over again by entering the “Enter” key, which will show the Main Menu scene again.

7 CONCLUSIONS

The aim of this thesis project was to be introduced to the game creating process without using any game engine. To achieve this goal, a video game was developed by only using DirectX, DirectX Tool Kit, and C++. A minimum viable product of the game was successfully implemented, and the game contains two levels, graphics, physics, head-up display, user interface, and gameplay features.

During the development process, the hardest part was learning the theories of the graphics pipeline and collision system. There was lots of linear algebra involved, and interactions with the graphics pipeline must be careful, since wrong inputs could result in unexpected results.

In conclusion, the project managed to achieve its objective. After the project, the programmer gained a better understanding of how things work and has enough foundation to tackle harder problems in game development.

7.1 Future work

Since the project is a minimum viable product, it has the potential to become a professional game. One aspect the game needs is sound. Another aspect is more gameplay features, for example, more pick-ups with different effects, a particle system to make things look more beautiful and immersive, and more levels with more enemy types to provide fun and challenging gameplay. Thus, this project has room to grow for future development.

REFERENCES

/1/ WePC. Video Game Industry Statistics, Trends and Data In 2022. Accessed 29.04.2022. https://www.wepc.com/news/video-game-statistics/

/2/ Wikipedia. C++. Accessed 29.04.2022. https://en.wikipedia.org/wiki/C%2B%2B

/3/ Wikipedia. DirectX. Accessed 29.04.2022. https://en.wikipedia.org/wiki/DirectX

/4/ Wikipedia. Visual Studio. Accessed 29.04.2022. https://en.wikipedia.org/wiki/Microsoft_Visual_Studio

/5/ Github. DirectXTK. Accessed 29.04.2022. https://github.com/Microsoft/DirectXTK

/6/ Learn OpenGL. Hello Triangle. Accessed 01.05.2022. https://learnopengl.com/Getting-started/Hello-Triangle

/7/ Vulkan Tutorial. Index buffer. Accessed 05.05.2022. https://vulkan-tutorial.com/Vertex_buffers/Index_buffer

/8/ Microsoft Official Docs. Introduction to Buffers in Direct3D 11. Accessed 12.05.2022. https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-resources-buffers-intro

/9/ Microsoft Official Docs. Graphics pipeline. Accessed 20.05.2022. https://docs.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-graphics-pipeline

/10/ Microsoft Official Docs. Vertex-shader stage. Accessed 27.05.2022. https://learn.microsoft.com/en-us/windows/win32/direct3d11/vertex-shader-stage

/11/ Learn OpenGL. Coordinate systems. Accessed 10.09.2022. https://learnopengl.com/Getting-started/Coordinate-Systems

/12/ Microsoft Official Docs. Pixel-shader stage. Accessed 17.09.2022. https://docs.microsoft.com/en-us/windows/win32/direct3d11/pixel-shader-stage

/13/ Ai, V. 2021. Texture Mapping in OpenGL. Accessed 30.09.2022. https://blogs.oregonstate.edu/learnfromscratch/2021/10/27/texture-mapping-in-opengl/

/14/ Learn OpenGL. Textures. Accessed 20.10.2022. https://learnopengl.com/Getting-started/Textures

/15/ Learn OpenGL. Transformations. Accessed 25.10.2022. https://learnopengl.com/Getting-started/Transformations

/16/ Game Programming Patterns. Game Loop. Accessed 30.10.2022. http://gameprogrammingpatterns.com/game-loop.html

/17/ DirectXTK Wiki. Github. Accessed 05.11.2022. https://github.com/microsoft/DirectXTK/wiki/Adding-the-DirectX-Tool-Kit

/18/ Microsoft Official Docs. Introduction to a Device in Direct3D 11. Accessed 10.11.2022. https://learn.microsoft.com/en-us/windows/win32/direct3d11/overviews-direct3d-11-devices-intro