Computer Graphics Report

Xiangyi Chen
u5752303

University of Warwick

December 20, 2025

Developed a C++ DirectX 12 third-person game with interactive characters, custom shaders, and data-driven levels demonstrating animation, collision, and advanced rendering techniques.

1 Introduction

1.1 Gameplay

The player controls a main character using W, A, S, D, with a third-person camera and mouse-controlled view direction, press '1' can release the mouse.

A cow NPC will approach and attack the player when nearby. The player can attack with the left mouse button to damage the cow. After the cow is knocked down, the player can press E to pick it up, carry it, and press E again to place it down.

1.2 Technologies

The game is developed in C++ using DirectX 12 with a modern GPU pipeline, including custom shaders, root signatures, PSOs, and resource management. It supports data-driven level loading through a custom file format.

The game features a third-person camera controlled by keyboard and mouse, basic AABB collision detection, and advanced rendering techniques such as normal mapping, parallax occlusion mapping, GPU instancing, alpha-tested meshes, vertex shader animation, and a simple skybox.

2 Technologies Implemented

2.1 Shader and Constant Buffer

A constant buffer stores frequently updated shader data. When the same shader is used across multiple draw calls, a ring buffer is used to avoid update conflicts. Each constant buffer has named variables, allowing updates by referencing the variable name.

```
1  class ConstantBuffer{
2  ID3D12Resource* constantBuffer;
3  unsigned char* buffer;
4  unsigned int cbSizeInBytes;
5  unsigned int maxDrawCalls;
6  unsigned int offsetIndex;
7
8  std::string name;
9  std::map<std::string, ConstantBufferVariable>
       constantBufferData;
10
11 void update(std::string name, void* data){
12     if (constantBufferData.find(name) !=
       constantBufferData.end()){
13         ConstantBufferVariable cbVariable =
       constantBufferData[name];
14         unsigned int offset = offsetIndex *
       cbSizeInBytes;
15         memcpy(&buffer[offset + cbVariable.
       offset], data, cbVariable.size);}}
```

The constant buffer deatils can be get from the shader file,so I make each shader control their onw constantbuffers.

```
1  class Shader{
2    std::string name;
3    ID3DBlob* shader;
4
5    //different types struct of constantbuffer
6    std::unordered_map<std::string, ConstantBuffer
       > constantBuffers;
7    ConstantBuffer_Manager* cb_manager;
```

The shader is initialized by loading the corresponding file and compiling it using the D3DCompile function.

```
1  hr = D3DCompile(shader_str.c_str(), strlen(
       shader_str.c_str()), NULL,
2      NULL, NULL, "VS", "vs_5_0", 0, 0, &shader, &
       status);
```

The reflect function can get the constantbuffer details and initialize them,

```
unsigned int reflect(Core* core){
ID3D12ShaderReflection* reflection;
D3DReflect(shader->GetBufferPointer(), shader->
    GetBufferSize(), IID_PPV_ARGS(&reflection));
D3D12_SHADER_DESC desc;
reflection->GetDesc(&desc);
    for (int i = 0; i < desc.ConstantBuffers; i
    ++){
        ConstantBuffer buffer;
        ...
        for (int j = 0; j < cbDesc.Variables; j
    ++){
            ConstantBufferVariable
    bufferVariable;
            bufferVariable.offset = vDesc.
    StartOffset;
            bufferVariable.size = vDesc.Size;
            ...
        }
        buffer.init(core, 1024);}
```

ShaderManager class control and store all shaders, when an object class need use a shader,then call the load function to init its shader, when updating the constantbuffer,just use the shader name, buffer name and variable name to update the specific data.

```
class Shader_Manager{
std::unordered_map<std::string, Shader> shaders;

void update(std::string shader_name, std::string
     cb_name, std::string var_name, void* data){
    if (shaders.find(shader_name) != shaders.end
    ()) {
        if (shaders[shader_name].constantBuffers
    .find(cb_name) != shaders[shader_name].
    constantBuffers.end()){
            shaders[shader_name].constantBuffers
    [cb_name].update(var_name, data);}}
```

2.2 One Textured and lit 3D Game level

A texture class stores basic material data, including the GPU resource and its descriptor heap index. It loads PNG files, uploads them to the GPU, and records the descriptor offset for later binding.

```
class Texture{
ID3D12Resource* tex;
int heapOffset;
void load(Core* core, std::string filename);
void upload(Core* core, int width, int height,
    int channals, const void* data);}
```

The Material class manages three textures (albedo, normal, rmax) for an object. Each material is identified by name, and all three PNG textures are loaded together by passing their file paths to the load function.

```
class Material{
  std::string name;
  Texture albedo; Texture normalmapping; Texture
     rmax;
  unsigned int heapoffset;
    void load(Core* core,std::string _name, std
    ::vector<std::string> filenames);
```

The heapoffset stores the location of the albedo resource within the descriptor heap. This is because the root signature of the resource is set to srvRange.NumDescriptors = 3,so the pixel shader can get the three resources from one-time 'SetGraphicsRootDescriptorTable' function.

```
//get the material first texture's gpu ptr
D3D12_GPU_DESCRIPTOR_HANDLE get_GPU_handle(Core*
     core){
    D3D12_GPU_DESCRIPTOR_HANDLE handle = core->
    srvHeap.gpuHandle;
    handle.ptr = handle.ptr + (UINT64)(
    heapoffset) * (UINT64)core->srvHeap.
    incrementSize;
    return handle;
}
```

All Materials are managed through the TextureManager class, which stores all Material data using a map of string and Material structures. The updateTexturePS function binds the selected material's texture descriptors to the pixel shader.

```
class Texture_Manager{
std::unordered_map<std::string, Material* >
    materials;
void updateTexturePS(Core* core, std::string
    material){
    core->getCommandList()->
    SetGraphicsRootDescriptorTable(2,
    get_material_GPU_handle(core, material));
}
}
```

The pixel shader samples the albedo and normal map using texture coordinates, reconstructs the normal in tangent space, and transforms it to world space using a TBN matrix. A simple diffuse light computed with a fixed light direction and combined with ambient lighting to get the final shaded color.

```
//simpe straight light
float3 LIGHT_DIRECTION = normalize(float3(0, -1,
     -1));
float3 LIGHT_COLOR = float3(1.0, 0.8, 1.0);
float3 AMBIENT_COLOR = float3(1.0, 1.0, 1.0);

float4 PS(PS_INPUT input) : SV_Target0{
float4 textureColor = tex.Sample(samplerLinear,
    input.TexCoords);
float3 mapNormal = normalsTexture.Sample(
    samplerLinear, input.TexCoords).xyz;

float3 tangentNormal = normalize(mapNormal * 2.0
     - 1.0);
//TBN
float3 normal = normalize(input.Normal);
float3 tangent = normalize(input.Tangent);
float3 binormal = normalize(cross(normal,
    tangent));
float3x3 TBN = float3x3(tangent, binormal,
    normal);

float3 worldNormal = normalize(mul(tangentNormal
    , TBN));

float diffuseIntensity = max(0, dot(worldNormal,
     -LIGHT_DIRECTION));

float3 ambient = AMBIENT_COLOR;
float3 diffuse = LIGHT_COLOR * diffuseIntensity;

float3 finalColor = textureColor.rgb * (ambient
    + diffuse);

return float4(finalColor, textureColor.a);}
```

Figure 1: *The side of a tree that is back side is darker than the other side, and the same applies to characters.*

2.3 Data Driven Level Loading

All model resource files are read through .gem files. The gem file stores all vertexs, bones, and animation information, the names of the required texture files. Using a for loop to initialize each mesh and animation instance.

```
void init_meshes(Core* core, std::string filename)
{
    GEMLoader::GEMModelLoader loader;
    std::vector<GEMLoader::GEMMesh> gemmeshes;
    GEMLoader::GEMAnimation gemanimation;
    loader.load(root, gemmeshes, gemanimation);
    for (int i = 0; i < gemmeshes.size(); i++) {
        Mesh* mesh = new Mesh();
        std::vector<ANIMATED_VERTEX> vertices;
        for (int j = 0; j < gemmeshes[i].verticesAnimated.size(); j++) {
            ANIMATED_VERTEX v;
            memcpy(&v, &gemmeshes[i].verticesAnimated[j], sizeof(ANIMATED_VERTEX));
            vertices.push_back(v);
            hitbox.local_aabb.expand(v.pos);
        }
        //use the albedo texture name as the matarial name
        textureFilenames.push_back(tex_root_alb);
        textures->load(core, tex_root_alb, filenames);
        mesh->init_animation(core, vertices, gemmeshes[i].indices);
        meshes.push_back(mesh);
    }
    //Bones
    memcpy(&animation.skeleton.globalInverse, &gemanimation.globalInverse, 16 * sizeof(float));
    for (int i = 0; i < gemanimation.bones.size(); i++){
        memcpy(&bone.offset, &gemanimation.bones[i].offset, 16 * sizeof(float));
        animation.skeleton.bones.push_back(bone);
    }
    //Animations
    for (int i = 0; i < gemanimation.animations.size(); i++){
        AnimationSequence aseq;
        aseq.ticksPerSecond = gemanimation.animations[i].ticksPerSecond;
        for (int n = 0; n < gemanimation.animations[i].frames.size(); n++){
            AnimationFrame frame;
            for (int index = 0; index < gemanimation.animations[i].frames[n].positions.size(); index++){
                memcpy(&p, &gemanimation.animations[i].frames[n].positions[index], sizeof(Vec3));
                frame.positions.push_back(p);
            }
            aseq.frames.push_back(frame);
        }
        animation.animations.insert({ name, aseq });
    }
    animation_instance.init(&animation, 0);}
```

I store map object transform matrices in .txt files. This avoids hard-coding large amounts of position data in the program and allows object transforms to be loaded during initialization.

```
bool load_instance_matrices( const std::string filename, std::vector<INSTANCE>& out_instances){
    std::string line;
    while (std::getline(file, line)){
        if (line.empty() || line[0] == '#')
            continue;
        std::stringstream ss(line);

        Matrix m;
        for (int r = 0; r < 4; ++r){
            for (int c = 0; c < 4; ++c){
                ss >> m.a[r][c];
            }
        }
        INSTANCE inst;
        inst.w = m;
        out_instances.push_back(inst);
    }
    return true;
}
```

Each line stores one matrix, with four data points per line.

- matrices
- 0.520521 0 0.196737 -1724.67 0 0.55646 0 0 -0.196737 0 0.520521 -854.003 0 0 0 1

2.4 Opaque Meshes and Meshes With Alpha Testing

This technique is implemented by whether or not the alpha test is called, when the texture's alpha is less than 0.5, it is discarded.

```
float4 PS(PS_INPUT input) : SV_Target0{
    if (textureColor.a < 0.5){
        discard;}
```

pictures here

Figure 2: *The transparent parts of the tree's leaves are using the alpha test.*

2.5 Vertex Shader Animation

The implementation of vertex shader animation requires changing the vertex position via time. I use a constant buffer that updates time data every frame.

```
cbuffer PerFrameBuffer : register(b2){
float time;
float3 cam_pos;};

PS_INPUT VS(VS_INPUT input){
    PS_INPUT output;
    float heightFactor = input.Pos.y;

    float wind = 0.0;
    wind += sin(time * 1.5 + input.Pos.x * 3.0)
    * 0.15;
    wind += sin(time * 2.0 + input.Pos.z * 2.0)
    * 0.1;
    float3 windOffset = float3(wind * 0.3, 0.0,
    wind * 0.2) * heightFactor;

    float4 worldPos = mul(float4(input.Pos.xyz +
     windOffset, 1.0), input.World); ...}
```

2.6 Instancing of Meshes

The instancing object uses a matrix vector to store all object positions. The input layout includes the world matrix. When calling 'draw', it sets a 'BufferView' array instead of the original view, and sets the number of instancing objects in the 'DrawIndexedInstanced' function.

```
class Object_Instance{
std::vector<INSTANCE> instances_matix;}

class Mesh_Istancing{
  ID3D12Resource* instanceBuffer;
  D3D12_VERTEX_BUFFER_VIEW instanceView;}

void Mesh_Istancing::draw(Core* core){
    D3D12_VERTEX_BUFFER_VIEW bufferViews[2];
    bufferViews[0] = vbView;
    bufferViews[1] = instanceView;
    core->getCommandList()->
    IASetPrimitiveTopology(
    D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    core->getCommandList()->IASetVertexBuffers
    (0, 2, bufferViews);
    core->getCommandList()->IASetIndexBuffer(&
    ibView);
    core->getCommandList()->DrawIndexedInstanced
    (numMeshIndices, numInstances, 0, 0, 0);
}
```

2.7 Third-Person Camera

The third-person camera move around a target character using yaw and pitch angles. Mouse input updates yaw and pitch, which are converted into a spherical offset to place the camera at a fixed distance from the target.

The camera always looks at the character with a vertical offset to keep it centered on screen. The final view and projection matrices are updated every frame to reflect the new camera transform.

```
class ThirdPersonCameraController{
Camera* camera;
Vec3* target_pos;     // charactor positino
float distance = 200.0f;        // distance to
    the charactor
void update_mouse(Window* window){
//mouse update judge
float dx = window->mousex - centerX;
float dy = window->mousey - centerY;

dx *= mouse_sensitivity;
dy *= mouse_sensitivity;

yaw -= dx;
pitch += dy;
pitch = clamp(pitch, -30.0f, 80.0f);
//reset mouse position back to the screen center
    .
}
void update_camera_transform(){
Vec3 target = *target_pos + target_offset;

float radiusYaw = yaw * M_PI / 180.0f;
float radiusPitch = pitch * M_PI / 180.0f;
Vec3 offset;
offset.x = distance * cos(radiusPitch) * cos(
    radiusYaw);
offset.z = distance * cos(radiusPitch) * sin(
    radiusYaw);
offset.y = distance * sin(radiusPitch);

camera->position = target + offset;

// Set the camera to look at the target point.
camera->target = target;

camera->forward = (camera->target - camera->
    position).Normalize();

Vec3 world_up = Vec3(0, 1, 0);
camera->right = camera->forward.Cross(world_up).
    Normalize();
camera->up = camera->right.Cross(camera->forward
    ).Normalize();

camera->pitch = pitch;
camera->yaw = yaw;
//update vp matirxs
camera->update_matrices();
}};
```

2.8 Animation Interaction

The 'MainCharactor' class controls the character's animations using 'ObjectAnimation'. Walking, running, or idlwchanges the 'movestate', while attacking or picking up items starts a timed action tracked by 'isdoingaction' and 'actiontimer'. When update checks player input, it updates which animation should play, and 'draw' function shows the animation on the character.

```
class Main_Charactor{
Object_Animation farmer;
//state data
Vec3 position;
Vec3 forward;
Vec3 up;
Vec3 right;
Matrix world_matrix;
Matrix hitbox_world_matrix;
//point the carrying object
Object_Animation* carrying_object;
//state control
bool is_carrying = false;
NPC_Base* carrying_item = nullptr;
bool is_doing_action = false;   // is attacking / grabing
float action_timer = 0.0f;
void handle_attack_input(){
...
if (window->mouseButtons[0] && !is_carrying) {
    is_doing_action = true;
    action_timer = 1.0f;
    move_state = Charactor_State::ATTACK_A;
}}
// check and reset the charactor state
void update_action(float dt){
    if (!is_doing_action)
        return;
    action_timer -= dt;
    if (action_timer <= 0.0f) {
        is_doing_action = false;
        current_animation_speed = 1.0f;
        //state after carrying
        if (is_carrying)
            move_state = Charactor_State::IDLE_WHEELBARROW;
        else
            move_state = Charactor_State::IDLE_BASIC_01;
    }
}

farmer.draw(core, world_matrix, camera->view_projection, ani_dt, move_state_helper[move_state]);
```

2.9 AABB

AABB class store the min and max position of boundbox,using these data to check if two hitboxes are intersect.(Figure 3)

```
class AABB {
Vec3 m_min;
Vec3 m_max;
inline static bool AABB_intersect(const AABB& a, const AABB& b){
if (a.m_max.x < b.m_min.x || a.m_min.x > b.m_max.x) return false;
if (a.m_max.y < b.m_min.y || a.m_min.y > b.m_max.y) return false;
if (a.m_max.z < b.m_min.z || a.m_min.z > b.m_max.z)return false;
return true;
}
```

Figure 3: *When there hitbox had intersect, NPC start attacking, charactor can also attack the NPC*

The HitBox class stores both the local (original) AABB and the world AABB, with the world AABB obtained by transforming the local AABB using the object's transformation matrix.The local AABB is initialized when init the meshes data.

```
class HitBox{
public:
  AABB local_aabb;
  AABB world_aabb;

void update_from_world(const Matrix& world){
    world_aabb = local_aabb.transform(world);
    world_aabb.update_cache();}
```

2.10 Parallax Occlusion Mapping

Parallax occlusion mapping needs the camera's and vertex's position, so I modify the PSIPUT structure add the world position.(Figure 4)

```
cbuffer PerFrameBuffer : register(b2){
    float time;
    float3 cam_pos;
};
struct PS_INPUT{
    float4 Pos : SV_POSITION;
    float4 Pos_w : TEXCOORD1;
    float3 Normal : NORMAL;
    float3 Tangent : TANGENT;
    float2 TexCoords : TEXCOORD;
};

float4 PS(PS_INPUT input) : SV_Target0{
//...TBN here
//Convert View direction into texture space
float3 viewDirWorld = normalize(cam_pos - input.Pos_w.xyz);
float3 viewDirTS = mul(viewDirWorld, transpose(TBN));
//Compute step size
float3 scaledViewDir = viewDirTS * HEIGHT_SCALE;
float stepSize = 1.0 / NUM_STEPS;
//Step through texture space until intersection
```

Figure 4: *The top one is POM, and the bottom one is normal mapping.*

Figure 5: *Different FPS when drawing different numbers of objects*

```
22 float2 currentTexCoords = input.TexCoords;
23 float currentHeight = 0;
24 for (int i = 0; i < NUM_STEPS; i++){
25     float sampledHeight = normalsTexture.Sample(
       samplerLinear, currentTexCoords).a;
26     if (sampledHeight > currentHeight)
27         break;
28     currentTexCoords -= scaledViewDir.xy *
       stepSize;
29     currentHeight += stepSize;
30 }
31 ...}
```

2.11 Skybox

The principle of a skybox is to generate a sphere and color it to the sky.Create the vertices of the sphere using the following method.

```
1  class Sphere{
2  void init_mesh(Core* core, std::string filename)
       {
3  std::vector<STATIC_VERTEX> vertices;
4  for (int lat = 0; lat <= rings; lat++) {
5      float theta = lat * M_PI / rings;
6      float sinTheta = sinf(theta);
7      float cosTheta = cosf(theta);
8      for (int lon = 0; lon <= segments; lon++) {
9      float phi = lon * 2.0f * M_PI / segments;
10     float sinPhi = sinf(phi);
11     float cosPhi = cosf(phi);
12     Vec3 position(radius * sinTheta * cosPhi,
       radius * cosTheta,
13         radius * sinTheta * sinPhi);
14     Vec3 normal = position.Normalize();
15     float tu = 1.0f - (float)lon / segments;
16     float tv = 1.0f - (float)lat / rings;
17     vertices.push_back(addVertex(position,
       normal, tu, tv));}}
18 std::vector<unsigned int> indices;
19 for (int lat = 0; lat < rings; lat++){
20     for (int lon = 0; lon < segments; lon++){
21     int current = lat * (segments + 1) + lon;
22     int next = current + segments + 1;
23     indices.push_back(current);
24     indices.push_back(next);
25     indices.push_back(current + 1);
26     indices.push_back(current + 1);
27     indices.push_back(next);
28     indices.push_back(next + 1);}}}
```

3 Evaluation

FPS is affected by the number of draw calls and the number of instancing objects. When I increase the number of objects, the frame rate is mostly stable above 100, but occasionally it drops below 100(Figure 5).

4 Limitaions

I tried to make the character move along the fence based on the direction of movement when it touches it, but I encountered some problems in calculating the movement method, which caused the character to get completely stuck and unable to move.

5 If Let me Do This Again

I try to refactor the structure of my classes. Every time the professor gives us a piece of code, I try to refactor it into what I think is a better structure. However, as the code grows, many structures become less concise and efficient.And maybe try more complicate technologies.

6 Conclusion

The project showcases real-time rendering, interactive gameplay, and efficient resource management, highlighting both achievements and areas for further optimization.

7 Links

GitHub:`https://github.com/Tabris-XiangyiChen/Computer_Graphic_Coursework`

Video Link: `https://drive.google.com/file/d/15GuNMz3bFBKQklGhGzejfYqbXu7IFdUo/view?usp=sharing`