Ubisoft Next 2024 (Angry Cows) Documentation

January 2024

Contents

Introduction

Welcome to the User Documentation of my project submission for the Ubisoft Next 2024 programming competition. This document breaks the down the API of the major subsystem for the engine I made as well as I made technical workings behind them
Some Features of the game engine includes

1. Multithreaded CPU SIMD 3D renderer accelerated with SIMD instructions with support for Vertex and Fragment Shaders
2. 2D Physics Engine with support for general concave polygons with support for general collision callbacks
3. Entity Component System to with tighly packed Components to exploits caches of modern hardware

Compilation

This project is written and compiled for Visual Studio Community 2022. Please compile the game in release mode for the best performance if you want to play the game.

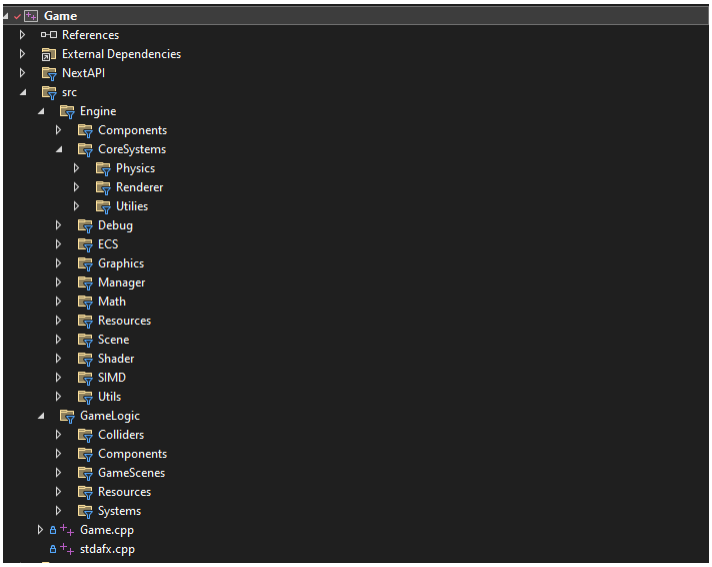
Warning!
This project uses AVX2 instruction set extensions for the x86 ISA to speed up the renderer which depending on your how old your CPU is, it might not support (Note: most intel/amd x86 processor in the last 10 years should support AVX2 so this shouldn't be a problem).

Just in case I have included a polyfill of the SIMD wrapper class that I wrote for this project. If you are getting an illegal instruction error when compiling, here are the steps to resolve the issue.

1. Go to project src directory and open SIMD.h
2. comment out the line #define SIMD_AVAILABLE

Code Structure

Below is the overview of the code structure when we first open the visual studio project



Here is the general the filter category mean

- **Engine**: Responsible For Rendering, Physics and Scene Management
  - **Components**: Component part of the Entity Component System
  - **CoreSystem**: Major Subsystems of the engine
    - **Physics**: 2d Physics Engine
    - **Renderer**: Rendering System as well as any system related to managing meshes
    - **Utilities**: Other minor core utility systems
  - **Debug**: Graphics used to debug the engine (not meant for external use)
  - **ECS**: Defines the Core Entity Component System Logic
  - **Graphics**: Define Game Asset class in the engine
  - **Manager**: Core Managment class
  - **Math**: Math library (Matrix, Quaternion, Vectors)
  - **Resources**: Extension of the ECS (explained further down in the documentation)
  - **Scene**: Defines abstract base class for all scenes in the game
  - **Shader**: Shaders for the Rendering System
  - **SIMD**: Wrapper around Intel SIMD intrinsics
  - **Utils**: Utility functions contains my homebrewed OBJ loader and other useful functions
- **GameLogic**
  - **Colliders**: Collider callback in the physics system
  - **Components**: Components in the ECS System
  - **GameScene**: All the different scenes in the game
  - **Resources**: Resource in the ECS System
  - **Systems**: System in the ECS System

Major System Breakdown

Entity Component System

The ECS System is the primary system which the developer would use to manage the Games State. The underlying ECS API is exposes by the ECSManager class. The ECSManager is a global Singleton instantiated on the startup of the game. Below we have code of a typical use case of the ECS System.

```
auto camera :shared_ptr<Camera> = ECS.GetResource<Camera>();
// Setup camera
camera->SetPosition( Vec3(0.0f, 0.0f, 0.0f),  Vec3(0.0f, 0.0f, 1.0f),  Vec3(0.0f, 1.0f, 0.0f));

for (int x = 0; x < 6; x++)
{
    for (int z = 0; z < 5; z++)
    {
        Entity meshEntity = ECS.CreateEntity();
        auto modelTransform = Transform(Vec3(x * 2, 0, 5 + z * 2), Quat( Vec3(1, 0, 0), 0.0));
        ECS.AddComponent<Transform>(meshEntity,  modelTransform);
        ECS.AddComponent<Mesh>(meshEntity, Mesh(Spot));
    }
}
```

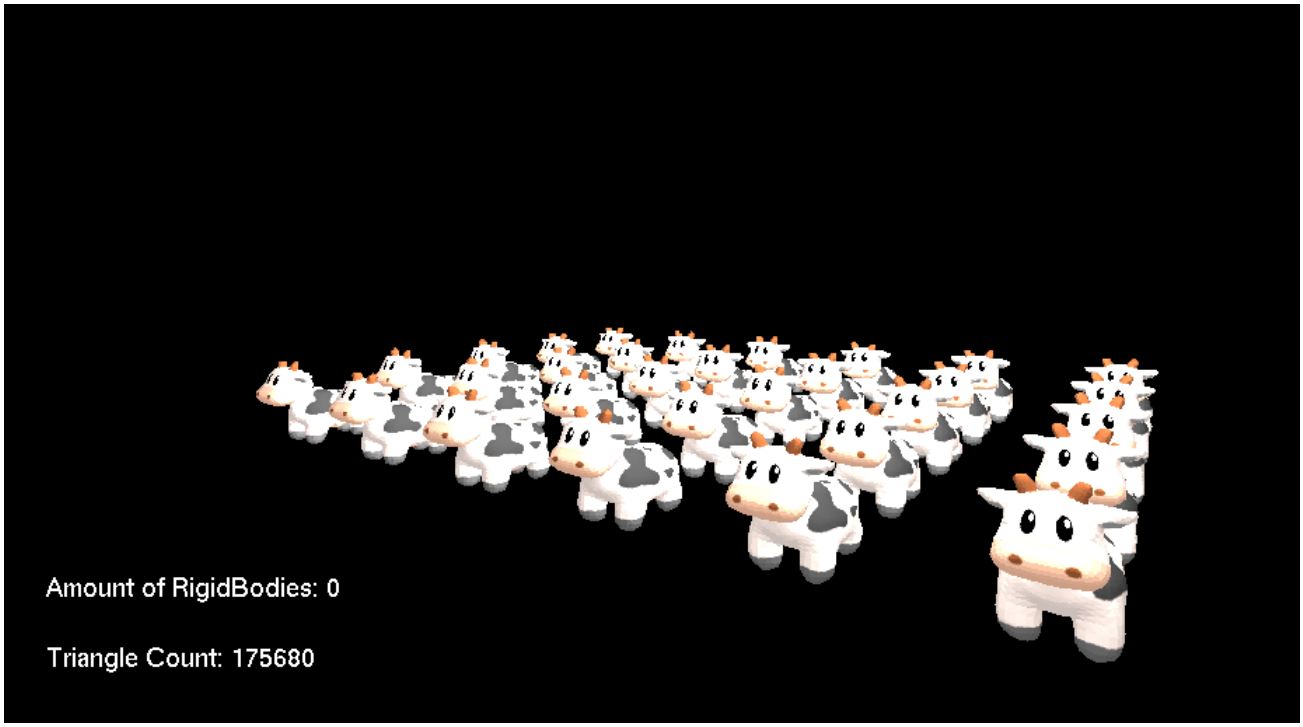
In the code above we're setting up the game scene on scene startup, on line 1 we see the line

ECS.GetResources<Camera>()

A Resource in the context of our ECS System is a globally unique singleton component. Not every data object can be mapped cleanly to an ECS systems and those that can't are abstracted into resources which any Systems can access. Our game engine does not allow for multiple Cameras so its a good candidate to be a resource.
Next we see the lines

```
Entity meshEntity = ECS.CreateEntity();
ECS.AddComponent<Transform>( meshEntity, modelTransform );
ECS.AddComponent<Mesh>(meshEntity, Mesh(Spot));
```

Here a Mesh Entity is created and two components a Mesh and Transform is attach to them. This is all that is required to render a mesh in our scene. Here is a picture of that scene for reference.



Next we would like to move and update our model enabling player control. These are the roles accomplished by Systems. There is no need to subclass any System base class as in the design of some other ECS systems. Systems in our engine can be any class or methods that calls on the Visit method in the ECSManager as illustrated below.

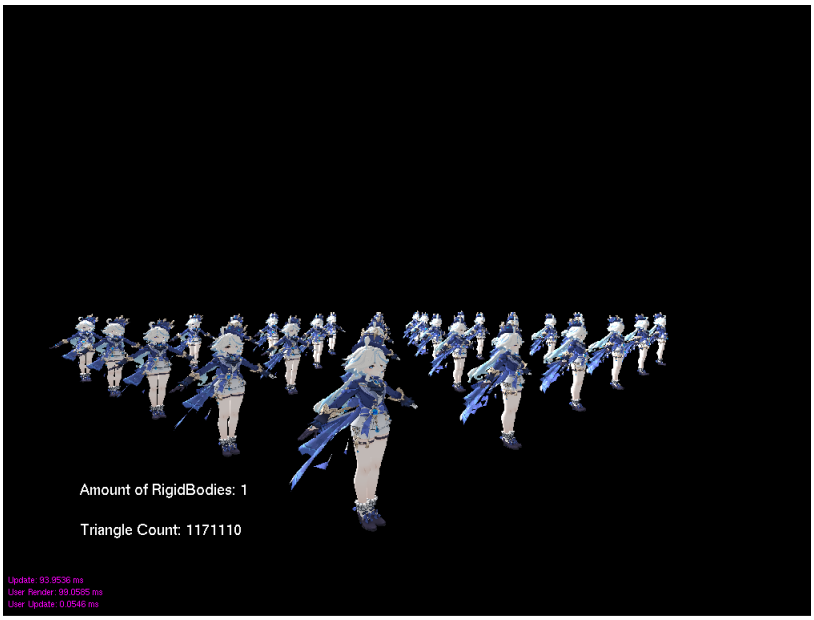
```
void DebugMesh::Update(float deltaTime)
{
    //entities.push_back(e);

    std::vector<Entity> entities;

    for (auto e:Entity : ECS.Visit<Mesh, Transform>())
    {
        auto& transform = ECS.GetComponent<Transform>(e);
        float speed = 1.0f * (deltaTime / 1000.0f);

        if (App::IsKeyPressed('Z'))
        {
            ECS.DestroyEntity(e);
        }

        if (App::IsKeyPressed('H'))
        {
            // move first entity -x
            transform.Update( { -speed, 0.0, 0.0 }, Quat());
        }


        if (App::IsKeyPressed('R'))
        {
            Quat rot = Quat( Vec3(0, 0, 1.0), 1.0 * (deltaTime / 1000.0f));
            transform.Update( { 0.0, 0.0, 0.0 }, rot);
        }
```

Above is just a System called DebugMesh we created we use an the ECS component method Visit to visit all meshes with components transform next we operate on the components updating it to advance our game state. Since systems are user define the user can control where they execute their System methods within the update function of scenes.

3D Renderer

Since I don't want to use the outdated OpenGL 1.0 API expose by underlying API I decided to write my own CPU renderer for the project. The renderer is tile based meaning it splits the screen into discrete tiles allowing for parrallelism when rendering and also avoids the use of any locks since no thread is at risk of writing to the same sets of pixels. Additionally AVX2 instructions are used to shade and rasterize sets 8 pixels to further speed up the renderer.

The renderer also implements deferred shading which means we don't shade pixel during rasterization but instead defer it until after the depth buffer is constructed this enable the renderer support many lights at once without as much of a performance hit. Although obviously it does not compare to modern GPU hardware it is noneless capable of performing impressive feats for a CPU based renderer. Below is an image of renderer rendering 1 million+ polygons at OKish 10fps.



To make things neater and avoid the costs of repeated loading a file from memory every time we load a 3d object. All 3d objects are store in the AssetServer class. At Setup of the scene a user would have to Load all the object they required to the assetserver and throughout the runtime of thhe scene they will be able to reference it through the AssetServer. To add a new object in our scene you must first add its path in the AssetServer method LookUpFilePath. You'll have to add another Enum to the ObjAsset list.

```
std::string LookUpFilePath(ObjAsset asset)
{
    switch (asset)
    {
    case Spot:
        return "./Assets/spotWithTextures.obj";
    case Furina:
        return "./Assets/furina.obj";
    case Maze:
        return "./Assets/MazeTriangulated.obj";
    case Pinball:
        return "./Assets/NoNormalsPinball.obj";
    case Pacman:
        return "./Assets/PacmanModel.obj";
    case Box:
        return "./Assets/BoxTest.obj";
    }
    assert(false && "not possible");
}
```

The rendering pipeline also has modern features such as Vertex Shader and Fragment Shaders. These shaders are customizable per object. Writing a Fragment Shader is as simple as subclassing the SIMDShader class let's take a look at it.

```cpp
class SIMDShader
{
public:
    virtual ~SIMDShader() = default;

    /**
     * \brief Abstract Shade Method called during fragment shading
     * \param pixel The sets of 8 pixels to shade
     * \param lights All lights in the scene
     * \param texture The texture belonging to the pixels
     * \param camera The current camera in the scene
     */
    virtual void Shade(SIMDPixel& pixel, std::vector<PointLight>& lights, Material& texture, Camera& camera) = 0;

};
```

Some default uniforms are already passed in to the shader whether a developer uses them is entirely optional. Also note this is where the SIMD nature of our rendering pipeline comes into play for developers. As you can see the shader doesn't recieve a single fragment it gets a SIMDPixel which combines the information of 8 pixels at once. So that means a SIMDPixel contains 8 pixel normals 8 worldspace position 8 UV coordinates etc etc. This means shading must be done entirely in SIMD instruction for max speed. I wrote a couple of thin wrappers around the SIMD instruction that are used to operated on these pixels. If you want to look at they are in the SIMD.h file. For examples of how some default Shaders are implemented please look at BlingPhong.cpp or ToonShaderSIMD.cpp under the Shaders filter.

As for Vertex Shading this is done in the VertexShader.cpp. This part isn't very extensible mostly because I didn't have any use for a vertex shader. But if you want to you can implement a vertex shading system exactly like how I implemented a the fragment shader system.

Physics System

Although the renderer is 3d the physics is actually 2d. The physics system at the beginning of the physics loop syncs up with the 3d position and orientation and at the end of the physics loop it forwards its new position and orientation to the 3d system to keep in sync.

The physics System is quite robust it is capable of supporting up to 1000 boxs at very little expensive to the processor. Additionally physics bodies can be assigned different collision categories. These collision categories can then be resolved by the player when two object of their specified categories collided.

```cpp
enum ColliderCategory
{
    Default,
    KillBox,
    Obstacle,
    Cow,
    Projectile
};
```

Above you can see the collider categories implemented for angry cows. Let's say we want the projectile to destroy the cows once it comes into contact with them. We simply subclass the collider class and implement the onCollide function illustrated below.

```cpp
class Collider
{
public:
    Collider(ColliderCategory A, ColliderCategory B) : m_Pair( {[&]A, [&]B}) {}

    virtual ~Collider() = default;

    virtual void OnCollide(
        Entity self,
        Entity other,
        RigidBody& selfRB,
        RigidBody& otherRB
    ) = 0;


    CollisionPair GetCollisionPair() const { return m_Pair; }

private:
    CollisionPair m_Pair;
};
```

```cpp
class ProjectileCowCollider : public Collider
{
public:

    ProjectileCowCollider() : Collider(Projectile, Cow) {}

    ~ProjectileCowCollider() override = default;

    void OnCollide(Entity self, Entity other, RigidBody& selfRB, RigidBody& otherRB) override;

};
```

Once we've implemented the collider function we register it with the ColliderCallbackSystem resource with the RegisterCallback and the physics system will take care of the rest.

Scene Management System

Scene management is very simple and easy with in our engine. Every Scene in the game must subclass the Scene class which provides the lifecycle hooks to allow the developer to interact with the game loop.

```cpp
class Scene
{
public:

    Scene() = default;

    virtual ~Scene() = default;

    /**
     * \brief Start is called once onm start up of this is used to register any
     *        Resources and setup any Systems a Scene might need
     */
    virtual void Start() = 0;

    /**
     * \brief Setup is called every time
     */
    virtual void Setup() = 0;

    /**
     * \brief calls every iteration of the game loop
     * \param deltaTime time in ms between this frame and the previous
     */
    virtual void Update(float deltaTime) = 0;


    /**
     * \brief called after pipeline render is done use this to render some
     *        custom graphics you need
     */
    virtual void Render() = 0;


};
```

At the beginning of the game all the scenes are registered into the GameManager and the active scene is set as the first scene that plays. GameManager itself is global instance whenever the developer wants to switch Scene they can call the same SetActiveScene to switch between scenes.

```cpp
void Init()
{
    // These must be initialized before any other systems
    ECS.Init();
    GameSceneManager.Setup();
    // Register all game scenes below
    std::unique_ptr<Scene> titleScreen = std::make_unique<TitleScreen>();
    std::unique_ptr<Scene> level1 = std::make_unique<Level1>();
    GameSceneManager.RegisterScene("Level1", std::move(level1));
    GameSceneManager.RegisterScene("TitleScreen", std::move(titleScreen));
    GameSceneManager.SetActiveScene("TitleScreen");
}
```

References

This what I used in and referenced in order to make this submission

1. Larrabee Rasterization
   This article talks about how do multi-threaded rasterization I referenced it alot when I was trying to build my tiled based renderer
2. Fast CPU Rasterizer
   This github repository contained alot good SIMD code applied to rasterization it taught me a lot about how to apply SIMD instructions to speed up code
3. ECS System
   The tutorial I followed when I was building the underlying ECS System for my game I have since heavily reworked it to make the API easier to work with and more performant
4. Code-It Yourself 3D Graphics
   I originally build a renderer based off this YouTube tutorial series before I rewrote my renderer to be multithreaded
5. How to Create a Custom 2D Physics Engine
   Tutorial for building a 2d engine I didn't reference it alot but it useful for structuring my physics engine
6. Separating Axis Theorem
   Tutorial explaining how Separating Axis Theorem worked and crucially how to determine the collision normal + penetrating depth
7. Contact Points
   The only good tutorial on how to find contact points after collision I found
8. Let's Make a Physics Engine
   Incredible Youtube Tutorial that did wonderful job of explaining dumbing down the math behind Physics Engine I referenced it alot when implementing rotations in my physics subsystem