Introduction to CSC8503 - Game Technologies

Introduction

Welcome to the Game Technologies module! In this module we're going to continue on from where Advanced Graphics for Games left off, by making some 3D interactive scens. To make them a little more interesting, we're going to look at ways of adding **physics** interactions between objects - applying *forces* to them to make them move, gravity to make them fall to the floor, and methods of detecting *collisions* between objects, and performing *collision resolution* so that both objects respond in a convincing manner. We'll also investigate ways of adding some **artificial intelligence** to objects, allowing them to make simple decisions about what to do via *state machines*, determine what they can see via *raycasting*, and also ways to navigate through the world via *pathfinding*, and take a look at how games can represent their world as either grids or even *navigation meshes*.

The codebase

Just as with the previous module, we're not going to program all of the supporting classes and functionality we need for these features from scratch; there's a small codebase containing things like vectors, matrices, and input device support. *Unlike* the previous module however, there's no need to write a 'renderer' class, as it has been provided for you, along with some additional classes to help you get started with the specific learning outcomes of the module. The code forms a small game framework, with a main game loop, and classes to interact with rendering, physics, AI, and pathfinding. Let's take a look at an overview of the major sections of the code!

The Renderer

There is a simple OpenGL-based renderer provided, that just as in the previous module, will handle all of the functionality behind getting images on screen. It is capable of drawing a set of objects on screen, that can be textured, and will cast shadows via shadow mapping. It assumes that everything will be drawn with the same provided shaders, so we don't even have to write any GLSL code in this module. It is worth examining the contents of renderer and shaders at least once though, as it may help reinforce the understanding of graphical rendering you gained in the previous module.

Debug drawing

One thing that this codebase does provide that you may not have seen before is the ability to draw simple lines and text to the screen, outside of the usual pattern of adding an object to the renderer and rendering it every frame. It's common in engines to allow transient 'debug' information to the screen that help you determine when things happen in your game, or what logic is being executed in a particular frame. As an example, the Unity game engine provides the **Debug.DrawLine** static function, that can be called anywhere, and will render coloured lines in the game world; this may be used to graphically show which player an enemy is targetting, or what nodes an AI player must travel through to get to their goal. We'll be using these functions later in the tutorial series to show the contact points between objects when they collide, or what state an AI opponent is in.

```
Vector3 a(0,0,0);
Vector3 b(0,100,0);
Vector4 pink(1,0,1,1);
Debug::DrawLine(a, b, pink);
```

Debug::DrawLine Example

GameObject

Rather than directly working with a scene graph in this tutorial series, we will instead use an array of *GameObjects* - these are a holding structure of other classes that represent interactions with specific parts of the codebase. To see what this means, let's take a look at the important member variables of the class:

```
namespace NCL {
   namespace CSC8503 {
      class GameObject  {
      public:
         //Various getters and setters
      protected:
         Transform          transform;

         CollisionVolume*   boundingVolume;
         PhysicsObject*     physicsObject;
         RenderObject*      renderObject;

         bool      isActive;
         string    name;
      }
   }
}
```

GameObject class pseudocode

We can give each object a *name* (handy for debugging purposes), and switch objects on and off with *isActive* - inactive objects essentially 'disappear' from rendering and physics interactions. More interesting are the *transform* variable which gives us the world position and size of a **GameObject**, and can give us both the local and world matrices we looked at in the previous module with scene graphs. There's an instance of a *collision volume* which allows us to define a simple shape for collision detection and visibility purposes (if you remember back in the previous module, we gave each scene node a 'bounding sphere' to test whether it was in the view frustum), as well as pointers to a **PhysicsObject**, and **RenderObject**. These provide the game object with data to allow the major systems of the codebase to interact with them, either by drawing it, or moving it in a physically accurate manner over time.

Transforms

Rendering and physics updates both need to know about where the object should be in the world, and which way it should be facing. In the previous module, we did this using a model matrix, but some things get a bit more tricky if we just have matrices. For example, we sometimes need to know an object's orientation, or an object's scale, but in matrix form these are both combined into the upper 3x3 region. To get around this, in this module we'll instead represent an object's spatial properties with a dedicated **Transform** class. Here's the relevant code details of the new class:

```
protected:
   void UpdateMatrix();

   Matrix4     matrix;

   Vector3     position;
   Vector3     scale;
   Quaternion  orientation;
```

Transform class member variables

To keep things simple, we're going to forego a scene graph for this module - we don't need to consider the placement of extra 'add on' meshes for our objects, and keeping track of the spatial relationship between two independently moving physics bodies is quite tricky. We will be looking at an alternative means of connecting objects together later on in the module, when *solvers* are covered.

By separating out the variables related specifically to object placement, we can then also keep separate the various logic we write for our objects - the code that is dedicated to rendering objects on screen doesn't care about what 'game' it is rendering, and should definitely not be able to influence the game logic. All it needs to know, is that there's something to be drawn, and it's at a specific transform in the world, provided via the matrix variable of our **Transform**, which will be kept up to date each frame by the game logic, by calling *UpdateMatrix*.

CollisionVolume

In the previous module, we gave our objects a graphical shape by loading in or generating 3D meshes. We also need to let the physics system know what shape our objects are, so that collisions between them can be detected and resolved. We generally don't use a shape as highly detailed as a 3D mesh, but instead a simpler representation, often called a bounding volume, or collision volume. A rocket fired by the player might have 1000s of polygons to graphcally represent it, but internal to the physics it may well just be a box, sphere, or cylinder - something that can be defined and tested against easily. To represent this in our code, we will make use of a number of subclasses of the **CollisionVolume** base class - spheres, axis aligned bounding boxes, and object oriented bounding boxes (don't worry if you don't know the difference, we'll be going over it when we get to collision detection!).

RenderObject

Just as functionality related to the spatial properties of an object were abstracted away to the **Transform** class, graphical properties have been moved to the **RenderObject** class. A quick look at the member variables should make this division clearer:

```
protected:
   MeshGeometry*  mesh;
   TextureBase*   texture;

   ShaderBase*    shader;
   Transform*     transform;
   Vector4        colour;
```

RenderObject class member variables

We have pointers to objects that represent a mesh, a texture, and a shader, as well as a reference to the object's transform (so the renderer can access the transformation matrices), and a **Vector4** to represent a colour - we'll be using this to show when an object has been clicked on, or collided with an object, and so on. As we're using a simple lighting model in the shaders, there's not a pointer to a bump map or gloss map, so there's not much to this class. In a more complex game engine, we'd probably not have the *texture* or *colour* variables at all, but instead a pointer to a 'material' - both Unreal and Unity represent the interface between a shader and how an object should be drawn as this conceptual material, which will include all of the uniform information that must be sent to a shader to correctly render the object.

PhysicsObject

Similar to how the functionality for drawing an object on screen has been encapsulated within a **RenderObject**, the code for this module also introduces a **PhysicsObject** class. As the name suggests, this separates out all of the logic and variables we'll need to allow objects to physically interact with each other, and apply forces to objects to make them move. Here's (some!) of the member variables of the class:

```
protected:
   float inverseMass;

   Vector3 linearVelocity;
   Vector3 force;

   Vector3 angularVelocity;
   Vector3 torque;
   Vector3 inverseInertia;
   Matrix3 inverseInteriaTensor;
```

PhysicsObject class member variables

If you've done any physics related work before, you may recognise some familiar concepts here, such as mass and velocity. If not, don't worry, we'll be covering what all of these variables mean, and how to use them, as we get to the relevant materials in the tutorials.

The GameWorld Class

To represent the overall state of the set of objects that make up our game / simulation, there's the **GameWorld** class. This lets us separate out the concept of 'a set of things that will interact with each other in some way' from the specifics of a particular game itself. Over the course of the tutorials, we'll be adding some variables and methods to this class, so let's have a look at how it is before we mess around with it:

```
namespace NCL {
   namespace CSC8503 {
      class GameWorld   {
      public:
         GameWorld();
         ~GameWorld();

         void Clear();       //Remove all objects
         void ClearAndErase(); //remove and delete all objects

         void AddGameObject(GameObject* o);
         void RemoveGameObject(GameObject* o);

         Camera* GetMainCamera() const {  return mainCamera;   }

         bool Raycast(Ray& r, RayCollision& closestCollision) const;

         void UpdateWorld(float dt, bool updateCamera);

         void GetObjectIterators(
         std::vector<GameObject*>::const_iterator& first,
         std::vector<GameObject*>::const_iterator& last) const;
      protected:
         void UpdateTransforms();

         std::vector<GameObject*> gameObjects;

         Camera* mainCamera;
      };
   }
}
```

GameWorld class header

There's methods for adding and removing **GameObjects**, as well as 'resetting' the entire gameworld. There's also an *Update* function that takes in a timestep, which performs a similar function to the *UpdateScene* method we used in the previous module. It will update the state of all of the **GameObjects** in the world (primarily to calculate new transformation matrices in the **UpdateTransforms** method), and provide a mechanism by which classes can iterate over all of the objects in the world - the renderer and physics system will use this to build up lists of objects to render, and objects to perform physics calculations on, respectively. There's also a flat list of **GameObjects**, and a pointer to a **Camera** instance, which works much the same as in the previous module.

As well as 'updating' the game world, we will later be adding methods to the class to allow information to be extracted from the current game state in order to make decisions about what to do with our simulation. For an example of this, cast your mind back to the previous module, where we performed frustum culling on the scene graph, to only draw the nodes that the camera could see. This could be encapsulated away from the rendering via a method that takes in a reference to a vector of **GameObjects**, and a **Camera** as parameters, and fills in the vector with all **GameObjects** within the game world that can be see by that camera - as well as rendering, its easy to imagine a case where maybe we want to know all of the objects within an AI opponent's field of vision, so keeping this functionality where it can be used in multiple different ways seems sensible. We'll see an example of extending the **GameWorld** class in the first proper tutorial, where we'll add in the ability to perform *raycasts* against the world, and work out which objects are hit by a straight line - handy for determining which object has been clicked on by the mouse.

The PhysicsSystem

The last of the new classes we'll be using encapsulates the functionality required to apply physics interactions between your objects. Here's (some) of it:

```cpp
namespace NCL {
   namespace CSC8503 {
      class PhysicsSystem  {
      public:
         PhysicsSystem(GameWorld& g);
         ~PhysicsSystem();

         void Update(float dt);
         void UseGravity(bool state) {applyGravity = state;}
      protected:
         void BasicCollisionDetection();
         void BroadPhase();
         void NarrowPhase();

         void ClearForces();

         void IntegrateAccel(float dt);
         void IntegrateVelocity(float dt);

         void UpdateConstraints(float dt);
         void UpdateCollisionList();

         GameWorld&  gameWorld;

         bool        applyGravity;
         Vector3     gravity;
      }
   }
}
```

PhysicsSystem class header

It holds a reference to the game world (so it can iterate over **GameObjects** and get the **Transforms** and **PhysicsObjects** out of them), and a selectable amount of gravity - we'll see how to add this into our game and make objects drop to the floor in later tutorials. There's an *Update* function, which takes in a timestep as normal, and we'll be adding methods to the class that will be called from within *Update* to detect collisions, resolve them, and split the world up using spatial acceleration structures to speed up certain physics calculations. There's a lot of protected methods in this class, so it *looks* like the **PhysicsSystem** will do a lot! And it certainly will...eventually! Most of those methods are actually empty in the code download, and you will be filling them in with the required calculations to make the physics engine work as the module goes on.

The TutorialGame

When the code is downloaded and executed, you won't get a grey screen like in the previous module, but instead something that looks like a full scene, with a selection of boxes and spheres floating in the air. Where do all these objects come from? To get you started with writing interesting simulations, you have also been provided with a very basic class to encapsulate what a 'game' might be built around the other provided classes. This **TutorialGame** class will initialise a set of meshes, textures, and shaders, and has a set of methods that will build up simple worlds comprising of **GameObjects**, that may have some combination of **PhysicsObjects** and **RenderObjects**. As an example, here's the method that will add a single sphere to the **GameWorld**:

```cpp
GameObject* TutorialGame::AddSphereToWorld(const Vector3& position,
       float radius, float inverseMass) {
   GameObject* sphere = new GameObject();

   Vector3 sphereSize = Vector3(radius, radius, radius);
   SphereVolume* volume = new SphereVolume(radius);
   sphere->SetBoundingVolume((CollisionVolume*)volume);
   sphere->GetTransform().SetWorldScale(sphereSize);
   sphere->GetTransform().SetWorldPosition(position);

   sphere->SetRenderObject(new RenderObject(&sphere->GetTransform(),
               sphereMesh, basicTex, basicShader));
   sphere->SetPhysicsObject(new PhysicsObject(&sphere->GetTransform(),
               sphere->GetBoundingVolume()));

   sphere->GetPhysicsObject()->SetInverseMass(inverseMass);
   sphere->GetPhysicsObject()->InitSphereInertia();

   world->AddGameObject(sphere);

   return sphere;
}
```

TutorialGame::AddSphereToWorld method example

This shows the basic flow of writing code that will create new **GameObjects** - if we want to see it, we need to add a RenderObject; while if we want it to collide with anything we need to give it a **BoundingVolume** and a **PhysicsObject**. The **TutorialGame** class has an *UpdateGame* method, designed to be ran from the **while** loop in our main function, much as with the Renderer methods we used in the previous module:

```
void TutorialGame::UpdateGame(float dt) {
	if (!inSelectionMode) {
		world->GetMainCamera()->UpdateCamera(dt);
	}

	UpdateKeys();

	SelectObject();
	MoveSelectedObject();

	world->UpdateWorld(dt);
	renderer->Update(dt);
	physics->Update(dt);

	Debug::FlushRenderables();
	renderer->Render();
}
```

PhysicsSystem class header

It's job is to make sure the camera moves properly, and that the physics is updated properly, before rendering a new frame to the screen. This class is designed to be later derived from with more game-specific logic - your coursework for example will probably be a subclass of this, and you are encouraged to use as much of the provided code as possible, as its all designed to help you understand how game code comes together, how classes keep things separated, and how to integrate the major components that will be described in this tutorial series.

Tutorials

As in the previous module, each day's learning outcome is taught via a tutorial, and they will build on each other, slowly improving the provided codebase into something that you could develop into a game proper. The tutorials are split into three areas (physics, AI, and networking) and we'll jump between these over the course of 3 weeks.

Week 1

For week 1, we'll be concentrating on getting some basic physics interactions working. We'll be starting by introducing the ability to use raycasting to the **GameWorld** class, to allow us to see what objects are under a mouse pointer, or whether object A can see object B. After that, we'll use the raycasting to select an object, which in the following two tutorials will have both linear and angular forces applied to it; once we have that, we'll then move on to how to detect and resolve collisions between objects that have begun to overlap while moving.

Week 2

With the basics of physics interactions in place, we can start to do some more interesting things in our game environment. We'll take a look at how to generate artificial intelligence via the use of state machines that manage what decisions an intelligent opponent can make, and what must change in the game world to make those decisions change. We'll also take a look at how to get those opponents moving around a maze in an intelligent manner via pathfinding, using the A* algorithm to generate the best possible path for the opponent to take.

Week 3

In the final week, we look at ways of improving upon the physics and AI systems. For physics, we'll investigate *solvers*, a way of forming more correct solutions to physics simulations, that allow for more accurate solutions that avoid accidentally injecting in too much energy that would otherwise make objects appear to jiggle about. We'll also see a more complex collision detection algorithm, that will allow us to use some more interesting collision volume shapes.

The Coursework

The coursework aims to bring all of the concepts taught on the module together, and allow you to experiment with changing the code around to best fit your gameplay ideas. Due to the shorter length of the module, the coursework is released on day 1, and you are encouraged to think about how each day's subject matter can be integrated into your coursework submission. On day 1, you should try to play about with the object spawning functions contained within the **TutorialGame** class, as they will allow you to rapidly prototype a few simple gameplay ideas. Investigating ways to reset the world and change the objects added to it will also provide useful, as this will let you create different game 'levels', so you can test bits of your code in isolation, and test gameplay ideas.