08356 ACW2

We're all Doomed

Game Engine

Nicholas Taylor

Matei Giurgiu

Nathan Glasper

Joshua Morley-Stone

Word Count: 1308

Contents

Changes to the initial framework

The framework provided was using an Entity-Component-System pattern which was later used in the engine and game development.

Changes to Entity

The Entity object has been added a few methods to enhance its core functionality. Firstly, a generic GetComponent method was added in order to make easy querying for a specific component and also it returns the specified type so no need for a cast.

```
ComponentGeometry geometryComponent = entity.GetComponent<ComponentGeometry>();
```

Figure 1: Usage of GetComponent method

We agreed that every Entity must have a Transform component by default. In the constructor when an Entity is created is automatically added a Transform component. For easy access this was stored in a variable transform. Therefore, querying for the transform component of an entity doesn't require using GetComponent method.

Entities now have a virtual Update method which can be overwritten by child objects in order to create behaviour. The Update will be called automatically by Behaviour System. In order for an Entity to have its Update called they first have to set *callUpdate* to true. This is similar to Unreal Engine's *bCanEverTick* variable.

They also have a virtual OnTriggerEnter method which is used by Collision System and its called when two trigger colliders overlap each other.

EntityManager now has a static method for creating/deleting objects at runtime. This was added because using the traditional EntityManager.AddEntity was changing the collection while iterating it (by the System Manager) and this is not allowed. My approach was using two static lists, one for object to be created and one for objects to be deleted. At the start of a frame and outside of any iteration of the entity list, entityManager.ManageEntities() is called which takes care of deleting/adding new objects.

```
EntityManager.Create(Bullet);
```

Figure 2: Creating entities at runtime.

Changes to Components

The initial example provided to the skeleton code was using a ComponentPosition was storing the position in the world for an Entity. This was changed later on to a more robust and complex **TransformComponent** which was able to store position, rotations, and scale. It uses Quaternions for storing and handling rotations which enabled us to handle more complex rotations very easy.

ComponentTexture was replaced by ComponentMaterial which also hold more data regarding how an object will be drawn. It contains a reference to a Shader object and also a tint colour.

Changes to Systems

The Render Systems now takes a Camera object in the constructor and a Light object. It uses the camera and the light object for drawing the scene. If the camera has attached a render texture (in the variable *activeRenderTexture*) the scene is rendered to that texture. This was used when creating the minimap because it required the scene to be drawn two times from different cameras and to

different targets. The parameters taken by the constructor help the engine code to get decoupled from the game code by providing an API and not having any reference to the Game class.

The Camera object originally designed used a full first person camera more like a modern FPS game in its design, later this was changed to match the specification of the ACW giving a more Doom inspired camera. The original camera was left present in the code just commented out for later use and review.

Another important change to ISystem is the addition of BeforeAction() method which is fired just before iterating through the entities list and calling OnAction on each one of them. The BeforeAction method is heavily used by the render system in order to clean-up the colour and depth buffer. It is also used in order to set the render target to the screen or to a render texture(*GL.BindFrameBuffer*).

Changes to Managers

Resource Manager no longer keeps the texture in an integer *Dictionary*. The textures now are managed as *Texture* objects and are stored appropriately. There is also another method for loading Shaders and a Dictionary for storing name. Since a shader is made up of two programs (vertex and fragment) a convention was created in which we pass only the prefix of the shader and the method will find the vertex and fragment using prefix + **"_vert.glsl"** and prefix + **"_frag.glsl".** Therefore, every shader will follow this structure (eg. *SimpleUI_frag.glsl* and *SimpleUI_vert.glsl* will be referred as *SimpleUI*)

Additions to the initial framework

A variety of new systems, components, and object have been added to the base framework in other to provide new features and modularity for the game code.

Components

1. ComponentCollider: abstract class for Box collider and Sphere collider which provides info about the collider shape and also provides methods for checking collisions.
2. ComponentSound: used for playing 3d sounds in the game.
3. ComponentUI: used by the UI Render System, it represents the UI elements which are drawn on the top of the game, provides data such as geometry, shader, texture and tint colour.

Systems

1. SystemBehaviour: responsible for calling the Update method on the entities which enabled this functionality (using *callUpdate* = true).
2. SystemCollision: this is doing the collision checks and calls the appropriate methods if the collision is of type trigger or does de-penetration to normal collisions.
3. SystemRenderUI: renders the entities which have ComponentUI attached.

Objects

1. Camera: stores the projection matrix, computes the view matrix from the attached transform, also stores clear colour and the active render texture if any. Used by SystemRender and SystemRenderUI to render the scene with.
2. CustomRandom: useful for generating random numbers or sequences of random numbers which don't repeat
3. Light: very basic light object which stores the direction of the light
4. Texture: base class for all types of textures used in the game
5. Texture2D: a texture which is loaded from a bitmap file

6. RenderTexture: texture generated on the GPU which serves as a render target when rendering
7. Shader: stores the OpenGL shader ID as well as shader variables locations, handy for having multiple shaders stored on a per object basis
8. Time: stores time in seconds since game started (useful for sinus-based effects) and delta time

Managers

1. New InputManager added in order to facilitate access to input elements such as mouse delta position of key presses without needing a reference.

```
if (InputManager.GetKeyDown(Key.W) && InputManager.GetKeyDown(Key.ShiftLeft))
```

Figure 3: Usage of InputManager

Game Entities Composition

	Transform	Material	Velocity	Collider_Box	Collider_Sphere	UI	Geometry	Cube	Sound
Environment	✓	✓	✗	✗	✗	✗	✓	✗	✗
Environment Wall Colliders	✓	✗	✗	✓	✗	✗	✗	✗	✗
Player	✓	✓	✓	✗	✓	✗	✓	✗	✓
Drone	✓	✓	✗	✗	✓	✗	✓	✗	✓
PickupItem	✓	✓	✗	✗	✓	✗	✓	✗	✓
BasicWeapon	✓	✓	✗	✗	✗	✗	✓	✗	✓
Bullet	✓	✓	✓	✗	✓	✗	✓	✗	✗
UI Hearth Icons	✓	✗	✗	✗	✗	✓	✗	✗	✗
UI Minimap	✓	✗	✗	✗	✗	✓	✗	✗	✗
UI Game Screens	✓	✗	✗	✗	✗	✓	✗	✗	✗

Engine Composition

The entire Engine codebase was decoupled from the main game logic. The main focus was developing and providing and API which can still send all the data the engine need while still keeping the game code pretty easy to use. The Engine.dll library contains the following items:

1. Components
    a. All the components were added in the Engine.dll
2. Objects
    a. Texture objects, Shaders, Time, Custom Random, Entity, Utilities, Camera and Audio were all added to the Engine.dll
3. Managers
    a. Every manager was also added to the Engine code, this includes: InputManager, Entity Manager, System Manager, Resource Manager
4. Systems
    a. Any system created was added to the Engine.dll library in the end.

The project with engine compiled can be found in the folder “Engine With Code Compiled”. The only source code present is the one used for creating the game related objects (eg. Player, Drone) the *MainEntry.cs* and the game window *MyGame.cs*

UML Diagrams

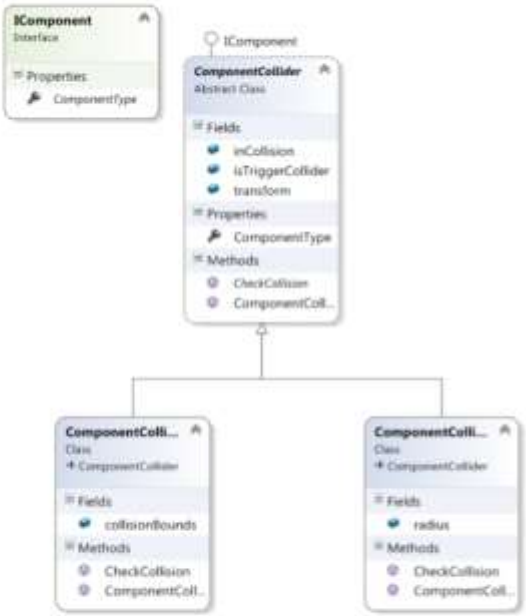

Figure 4 Collision System

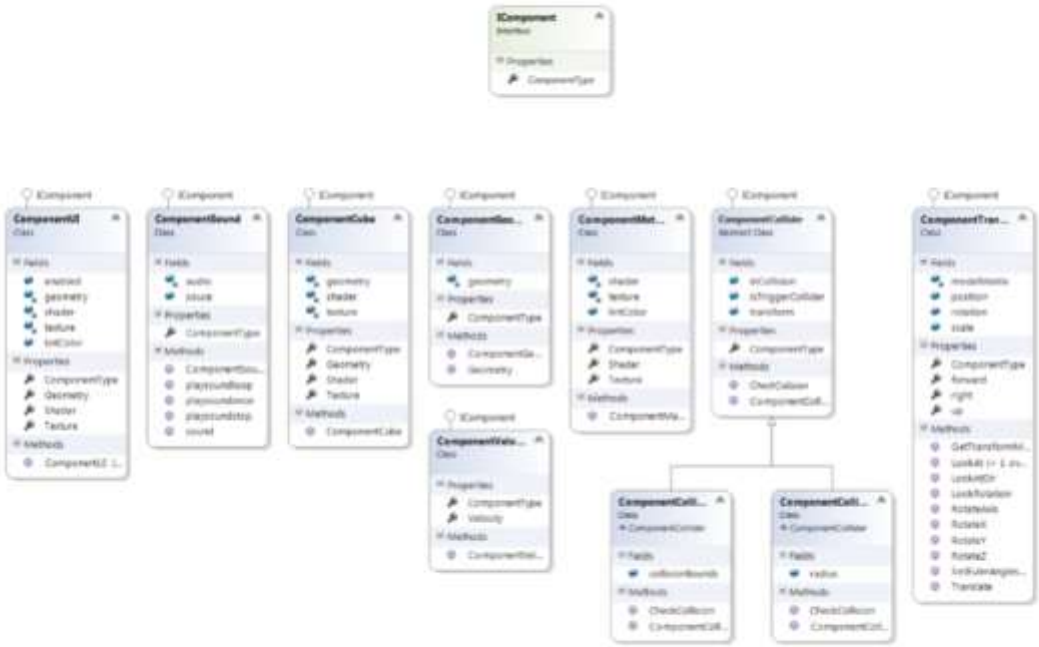

Figure 5 Components

Figure 6 Entities

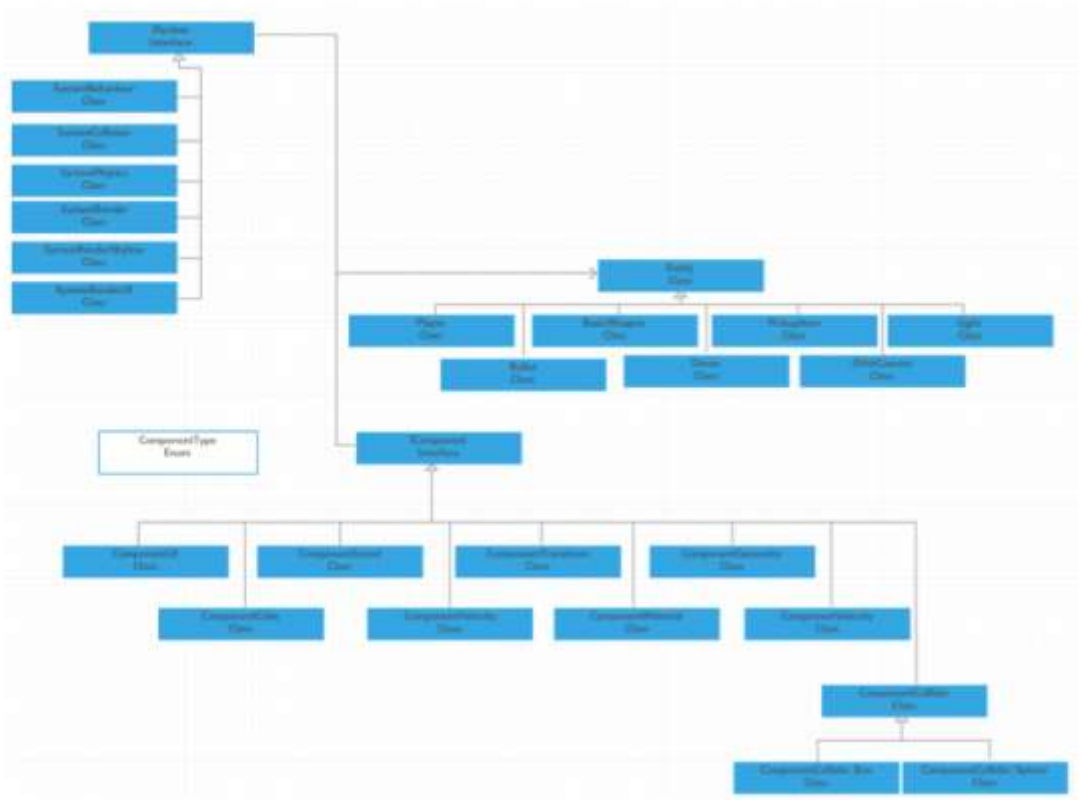

Figure 7 Entity System Component

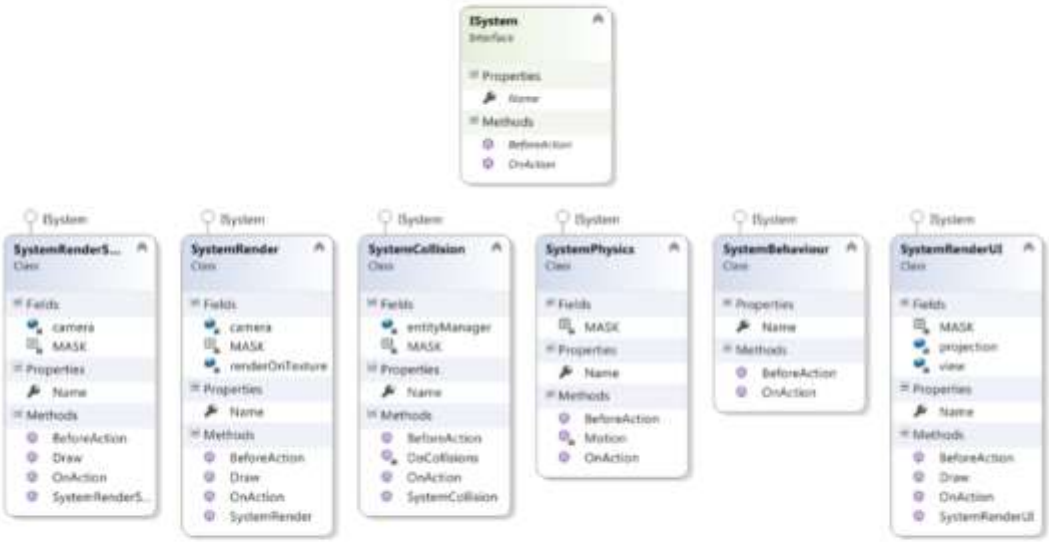

Figure 8 Systems

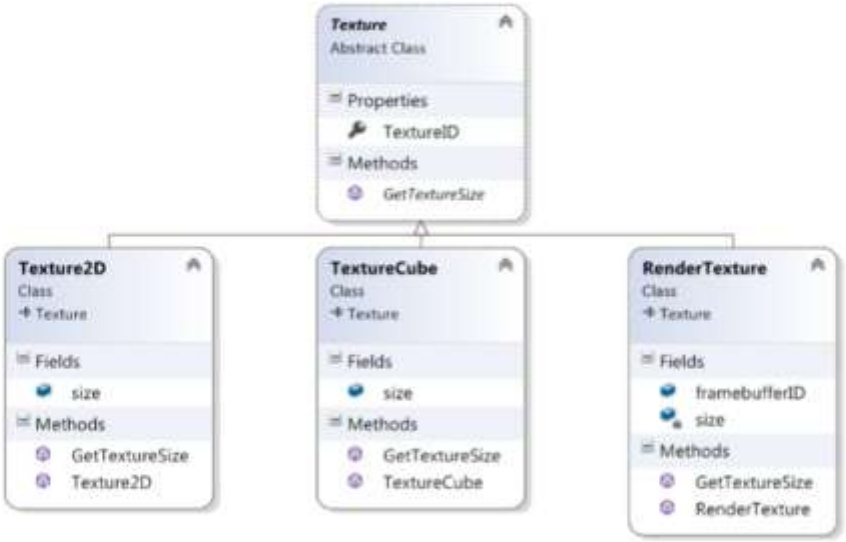

Figure 9 Texture Object