Game Engine Programming
Lab 2: The Component Entity System

Karsten Pedersen
Department of Creative Technology

October 7, 2019

In this lab we are going to be focusing on the core design of our engine. This actually means that we wont be touching OpenGL (or DirectX) just yet. Instead we want to look at creating an easy and flexible architecture which will stand the test of time and hopefully ensure that yourself or anyone using your engine to make an actual game will have a pleasant experience. Some factors which detract from this include:

- Large amounts of code required for simple tasks
- Slow compile times
- Inflexbile
- Awkward or unclear API
- Too easy to make errors

As we go through this series of labs, we will cover a number of ways to avoid these problems. First lets look at a number of other engines and see how they work.
Lets say a task we want to achieve is to add a simple cube to the scene at a certain position and rotation. The first example we will use is Unreal Engine 4:

```
// Create actor at specific location
FActorSpawnParameters si;
GetWorld()->SpawnActor<AActor>(FVector(), FRotator(0.0f, 45.0f, 0.0f), si);

// Create mesh
UStaticMeshComponent* mesh =
  CreateDefaultSubobject<UStaticMeshComponent>(TEXT("Cube Mesh"));

// Assign it to the actor
actor->SetRootComponent(mesh);
```

In general, the process is quite straight forward, however some of the types are a little awkward to remember and quite long to type such as *FActorSpawnParameters*, *UStaticMeshComponent* and *CreateDefaultSubobject*. The *U*, *A* and *F* prefixes are part of the coding standard which can be a little confusing. The requirement of a *FActorSpawnParameters* reference is also quite verbose. The *TEXT* MACRO is also easy to forget unless you come from a C background.
Now there are a number of technical reasons for this design such as avoiding namespaces, unicode and even to avoid *std::string* because in the early days, this was still a little bit too young or expensive in terms of performance. Some of these issues you might want to find other solutions to in your design.

Note:
At first glance one of the largest issues you might see with UE4 is that it is using *raw pointers* for *C++* classes. This is extremely bad coding practice. However, if you did a little deeper, you will see that the entire memory management system of UE4 is actually based on *Garbage Collection* and *Virtual Memory* providing a good amount of safety at the expense of a little bit of performance and portability to otehr platforms. Whilst I would not advise using this for your engine, you might like to read about this more:

- `https://wiki.unrealengine.com/index.php?title=Garbage_Collection_%26_Dynamic_Memory_Allocation`
- `http://man7.org/linux/man-pages/man2/mprotect.2.html`
- `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366898(v=vs.85).aspx`
- `http://www.hboehm.info/gc/`

The second example is Unity. You should see quite a difference here because unlike UE4 which has evolved as needed by Epic Games to support their requirements internally (whilst trying hard to avoid it becoming a sprawling mess!), Unity has been engineered from the ground up to be an easy to use prosumer engine. This has lead to a much more straight forward API to those less familiar with 3D software evelopment.

```
// Create GameObject and set location
GameObject go = new GameObject();
go->GetComponent<Transform>().eulerAngles = new Vector3(0, 45.0f, 0);

// Create a mesh
Mesh mesh = Resources.Load<Mesh>("CubeMesh");

// Add a mesh renderer
MeshRenderer mr = go.AddComponent<MeshRenderer>();

// Add a mesh filter and assign our loaded mesh
MeshFilter mf = go.AddComponent<MeshFilter>();
mf.mesh = mesh;
```

Some of the most immediate areas of interest is the use of public variables to set (i.e *eularAngles*). These are likely to be using the properties feature of C# rather than *setters* and *getters* which can sometimes lead to more expensive code being executed than the developer first assumed. Perhaps the biggest issue to note is that a few more lines are needed to achieve the same result as in UE4 due to the addition of a *MeshFilter*. You might at first wonder why not just assign the *Mesh* directly to the *MeshRenderer*? The main reason I suspect they have done this is to further separate out the components for better use within their editor. Writing code to generate objects in this way is not the most recommended method of doing things within the world of Unity, instead dragging and dropping within the editor is preferred. This unfortunately leads to quite a bit more code required to perform simple tasks.

So now it is our turn. Since we don't have the baggage of legacy requirements, the complex requirement of fun-to-use tools or scripting languages, we can attempt to make our API as simple to use and as efficient as possible whilst still retaining a powerful feature set.

Note:
At this point you may want to have a lok at some of the other popular engines in use today such as *CryEngine*, *Source Engine* and *Godot*. This kind of research will not only be good for your assignment but will also help provide a good showcase of what works well and what could potentially become a maintenance nightmare at the end. Perhaps attempt the same example of adding a cube to the scene using pure code and try to compare the differences.

The following is an example API that you might decide upon. I basically came up with this by looking around at other engines whilst playing around in a text editor and a UML diagram tool until I found something that I wasn't going to get sick of using throughout this unit. So I suggest you do the same instead of just copying this code.

```
#include <myengine/myengine.h>

#include <memory>

using namespace myengine;

int main()
{
  // Initialize our engine
  std::shared_ptr<Core> core = Core::initialize();

  // Create a single in-game object
  std::shared_ptr<Entity> entity = core->addEntity();

  // Add a very simple component to it
  std::shared_ptr<TestScreen> testScreen = entity->addComponent<TestScreen>();

  // Start the engine's main loop
  core->start();

  return 0;
}
```

Ignoring the fact that in this example we have also needed to include the headers, add a *main* function and started up the engine, you will see that there are only two lines needed to add an *Entity* and an example *TestScreen* to it.

Smart pointers and RAII provides *C++* with fantastic (and quite unique) functionality when dealing with memory, however it is a little bit long winded to type. We can address this by using *typedefs* or *MACROS*. The simplest way would be to define in your engine's headers the following:

```
#include <memory>

#define shared std::shared_ptr
#define weak std::weak_ptr
```

Now in your code, it would more look something like:

```
// Create a single in-game object
shared<Entity> entity = core->addEntity();

// Add a very simple component to it
weak<TestScreen> testScreen = entity->addComponent<TestScreen>();
```

Note:
The *MACRO* or *typedef* approach will be great to see in your **game** to demonstrate the use of the engine but I might suggest that it will be easier for me to follow your code if in your **engine** you stick to using the full name, i.e *std::shared_ptr*. Remember, they should end up completely interchangable anyway.

So now I am going to leave you to it to explore and come up with your own engine design you are happy with. I have added a UML diagram describing the approach that I am going to take in **Figure 1**, yours might end up being similar but certainly does not have to if you have a good reason for it.

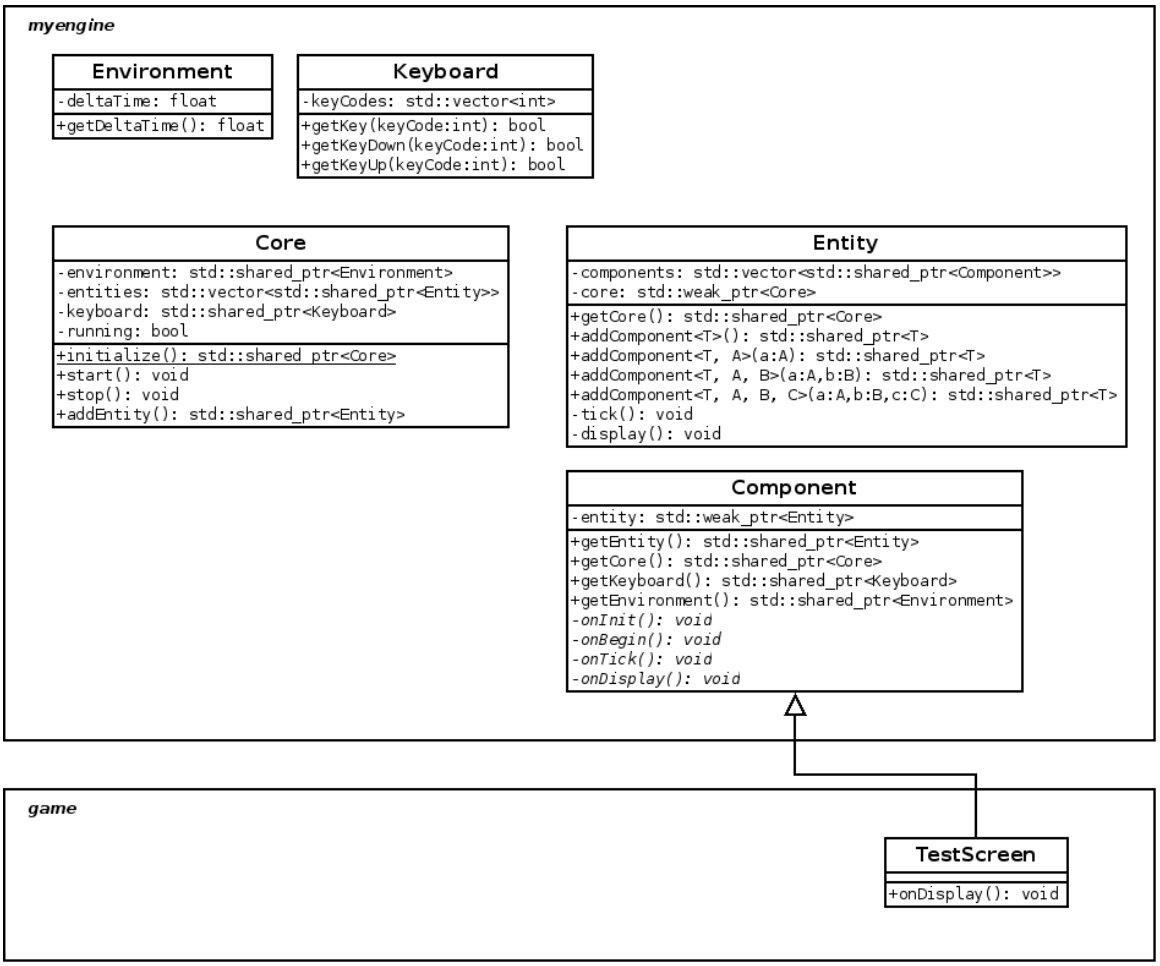

Figure 1: *The structure of an early game engine*

A small number of points that you might want to think about:

- Why do I require *core::addEntity* rather than *new Entity*?
- Why do I avoid static classes like *Application, Environment, Keyboard* and instead have *getters* in the *Component* class?
- Why do I have multiple *Entity::addComponent* template functions?