Toffee Engine Manual

Engine overview

The toffee engine is a 3D game engine with the functionality to support a variety of genres. Perhaps best suited to small arcade style games, with a little more development on the engine side of things it could feasibly support a first-person shooter like Counter-Strike or Call of Duty (the traditional multiplayer mode with smallish enclosed maps, that is. Making Warzone would be a bit of a stretch).

As the engine is for demonstration purposes, many of its features aren't fully fleshed out (though easily could be in the future).

The engine currently only runs on Windows, but consideration has been given to extensibility so that other platforms can be supported in the future.

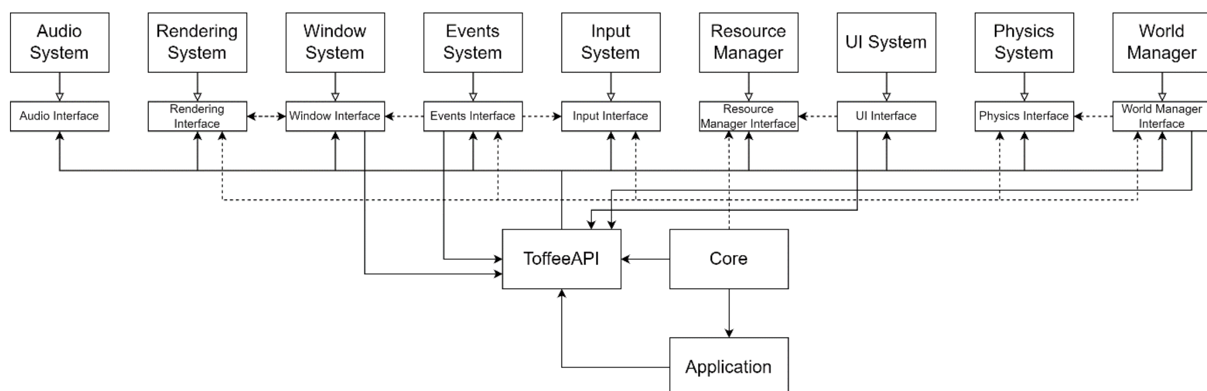
The main features and libraries of the engine:

- 3D rendering with fixed-function OpenGL 1.1 – already pre-installed on most systems.
- Real-time 3D physics simulation with Nvidia PhysX version 5.3.0.33308187
- Audio with SoLoud version 20200207
- Events and Window management with SDL version 2.28.1 . This is also how the OpenGL context is created.
- User input with Gainput version 1.0.0
- Graphical UI making use of OpenGL and SDL_ttf version 2.20.2
- Entity component system with EnTT version 3.13.0 for managing game objects and scenes.
- Resource management uses Minizip version 1.1 and boost version 1.84.0 .

The engine has a core system which runs the game loop and is where calls are made to the application. The loop works as follows :

```
While true {
        Process events
        Update subsystems
        Update application
        Render
}
```

The application is an object owned by Core through a unique pointer. All the subsystems are interfaced with through a service locator as described in Game Programming Patterns (Nystrom, 2014). This is a static class ToffeeAPI which holds a unique pointer for each subsystem, and is accessible anywhere by including it's header file. This diagram attempts to describe the system:



Solid lines show directional association, e.g. the core needs to know about the application object.
Dotted lines show friendship e.g. the world manager needs to be able to call a private physics function but doesn't need to know about the physics object itself. It accesses the physics through the ToffeeAPI like anyone else would. The need for friendships for accessing private functions is because if everything was public, the

application programmer (or anything else) could access them, and if you don't know what you're doing, well, fudge.
Note from the diagram that for the most part, the actual implementations of the various subsystems don't need to know about each other. Swapping in and out subsystem implementations is therefore theoretically only slightly painful, as opposed to nightmarish. Also note that the service locator is the only thing the application knows about.

Setup

In the Toffee folder there are two subfolders along with the Visual Studio solution Toffee.sln – open this up. The solution is comprised of two projects:

- Engine – This controls the programs executable and all core systems and subsystems.
- Game – this is where the game logic is defined. It builds as a static library which is then included in the engine.

Make sure Engine is set as the startup project. The executable will be output in Engine/x64/Debug . **There are six DLLs which must be placed alongside the executable in this folder**, which are included in Engine/lib . These DLLs are:

- PhysX_64.dll
- PhysXCommon_64.dll
- PhysXFoundation_64.dll
- SDL2.dll
- SDL2_image.dll
- SDL_ttf.dll

If successful you should see a scene, with text giving instructions on how to use the demo. Enjoy fiddling with it.

Getting Started

To start building your own application, you may modify the demo files or replace them with your own. Your main header file must #include ToffeeEngine.h, your core class must inherit from Toffee::App::Application, and must implement two virtual functions, Start and Update. There is also a required function definition for `Toffee::App::CreateApp`. A full header file template:

```
#pragma once
#include <ToffeeEngine.h>

class Game : public Toffee::App::Application {
public:
	// Required by the engine. Perform any initializations here.
	void Start() override final;

	// Required by the engine. Called every frame, your game logic goes here.
	///<param name="dt">Time since the last Update</param>
	void Update(double dt) override final;

private:
	// Your own functions and member variables should go here.
};

//Create the application layer
//See Application.h
Toffee::App::Application* Toffee::App::CreateApp(int argc, char** argv) {
	return new Game();
}
```

As discussed earlier, you have one interface to the engine, `Toffee::ToffeeAPI`. An example of using this to play a sound: `Toffee::ToffeeAPI::Audio()->Play("MySound");`
Clearly this is already quite lengthy so you may want to define a typedef as shown in the demo code:

```
typedef Toffee::ToffeeAPI API;
```

Let's draw a cube. To begin, you'll need to open a window and create a new scene:

```
API::Window()->OpenWindow("Window Title", 1280, 720);
API::World()->SwitchScene("MyScene");
```

Now all you need to do is create a new game object and attach a cube mesh component to it:

```
Toffee::GameObject cube = API::World()->CreateGameObject(-2.5, -1.625, 0);
API::World()->SetMesh(cube, "Cube");
```

This should leave you with an untextured cube on screen! Toffee provides four built-in meshes: Cube, Plane, Quad and Sphere. Toffee also supports loading of .obj files for custom models.

To load models, textures, audio etc, you'll need to mount at least one real location to a virtual location. The real location can be a local folder or in a .toff file (really just a .zip file). Toffee supports loading asynchronously!

Supported file formats:

txt wav, mp3, obj, png, jpg,.jpeg, ttf

The resource system is designed for ease of use – just give a location and a name for the resource, Toffee will work out what it's for. When using a resource you only need to know its name. For example:

```
API::Resources()->Load("Resources/ball.png", "BallTex");
// ...
API::World()->SetTexture(player_, "BallTex");
```

Issues and Quirks

1. When performing various operations such as adding components to game objects, the order you do it sometimes matters. If you're doing something wrong, the console will tell you.
2. Physics collisions – sometimes the ball can penetrate through the end of the crate for a moment, perhaps this is a PhysX bug though it may be a logic error on Toffee's part.

Generated with AI at fotor.com