# Game Engine Architecture

## Homework 1

### Due to: 23:55, 20.04.2017

We talked about how an effective game object model is based on an entity-component system which lets users create complicated object behaviours using a preset of reusable components. It's time to implement one now!

### Task

We require that you design and implement an entity-component system which lets users define new objects using a text-based file format. The concrete implementation and design choices are left to you but we greatly encourage you to take time into creating a sensible system that you'd actually enjoy using.

### Requirements

- Your system must implement the following interface:

```
class IWorld
{
    public:
        /// Reads the file specified by file
        /// and prepares the system for later calls to SpawnObject.
        /// All types described in the file must be spawnable.
        virtual void ParseTypes(const char* file) = 0;
        /// Spawns a new object of the given type with the given name.
        virtual void SpawnObject(const char* objectType, const char* objectName) = 0;
        /// Updates the world and all of its objects.
        /// Logs any information to the output file.
        virtual void Update(float deltaMs) = 0;
        /// Destroys the world and all internal types, freeing all the memory.
        /// In its simplest form, it's just Destroy() { delete this; }
        virtual void Destroy() = 0;
};
/// Creates a new world. Remember PIMPL?
/// In its simplest, it's just CreateWorld() { return new World(); }
extern "C" IWorld* CreateWorld();
```

- We require objects to have a name and position (although we won't interact directly with that so you can decide on how to store them)
- The expected input file format is as follows:

```
class <classname1>
- <componentType11>
- <componentType12>
- <componentType13>
class <classname2>
- <componentType21>
- <componentType22>
- <componentType23>
...
```

- Example:

```
class Mario
- Jumpable(20, 1, 5)
- Renderable
...
```

- `IWorld::SpawnObject` must create a new object in your internal collection.
- `IWorld::ParseTypes` must parse all types.
- We guarantee that the input file will be be in format *<name>.in* and we expect the output of your library to be *<name>.out*
- When `IWorld::Update` is called, your system must update all spawned objects and all components. Updating the system must print the following message to the output file:
  *--- Starting frame <N> ---*
  where <N> is the current frame index. The first frame has index 0.
- Objects never die.
- Components are updated in the order they are listed in the file
- Renderable will always be listed last
- Objects are updated in the order they are spawned.
- The list of components is predefined and we expect you to have the following components whose Update methods behave as follows

| ***Component*** | ***Behaviour on Update*** |
|---|---|
| Renderable | Prints `<objectName> is located at (<x>, <y>)` to the output file each frame. This should be the last component to be executed. |
| Jumpable(height, time, delay) | For the next `time` seconds moves the object vertically `height` units up; after that `time` seconds have passed it moves the object vertically `height` units down, until it reaches height 0 again. There's a delay of `delay` seconds between jumps. There is no delay |

| | |
|---|---|
| | between spawning the object and the first jump i.e. the object starts jumping as soon as it's spawned.<br><br>Example for Jumpable(20, 1, 5):<br>If an object starts at (0, 0) and the jump begins immediately. 1 second after the jump has begun, the object should be at (0, 20), and after another second it should be at (0, 0) again. For the next 5 seconds nothing happens and after that the process is repeated. |
| FibonacciWalk(maxFibIndex, sleepInterval, sleepDuration) | The object must walk along the abscissa with distance `(frameIndex % maxFibIndex)`-th number of Fibonacci each frame where `frameIndex` is the index of the current frame. Consider 1,1 to be fib(0) and fib(1). Every `sleepInterval` seconds the object falls asleep and won't move for the next `sleepDuration` seconds. This movement is independent of actual time passed, only the frame index is taken into account, however the sleep-related parameters are affected by the delta time. `frameIndex` is a global counter - starts from 0 during the first call to `World::Update` and increases by 1 for frame. The frame index is the same for all objects in the world.<br><br>Example for FibonacciWalk(10, 20, 30):<br>If an object starts at (0, 0) at frame 0, it should move with distance of `fib(0%10)=1` steps to the right and end at (1, 0). At frame 1, it should move with speed `fib(1%10)=1` steps to the right and end at (2, 0). At frame 55, it should move with speed `fib(55%10)=8` steps to the right. Every 20 seconds, the object must fall asleep and skip movement for the next 30 seconds. |
| Shooter(Projectile, interval) | This component shoots a projectile every `interval` seconds by spawning a new object of the type `Projectile`. Projectile is guaranteed to be a type described in the input file. When spawning the object the following message should be logged: '<shooterObject> just shot <projectileName> - a projectile of type <Projectile>' where `<shooterObject>` is the name of the current object, `<projectileName>` is the name of the spawned projectile and `<Projectile>` is the name of the projectile type. The name of the spawned projectile should be in the form <Projectile><Counter> where <Counter> is increased separately for every projectile (e.g. the name of an object of type Spell should be Spell1, Spell2, Spell123123).The spawned new object acts as described in its own behaviour.<br><br>Example for Shooter(Bullet, 5) attached to an object with name Mario: Every 5 seconds, the component will spawn a new Bullet object (say 'Bullet1' is the name of one of the objects) and log 'Mario just shot Bullet1 - a projectile of type Bullet'. 'Bullet1' is updated along with the other alive objects. |

| | |
|---|---|
| Multiplier(mushroom Locations, triggerDistance) | This component takes the `mushroomLocations` parameter which is an array of 2D vectors, e.g. an example instantiation would look like Multiplier([(0, 0), (1,1)], 1). If by moving (due to Jumper / FibWalk) the owner object of this component gets to distance less than `triggerDistance` to any of the `mushroomLocations` that mushroom explodes and multiplies all the non-sleep related numeric parameters on other components by 2. That would be the `maxFibIndex` & `maxHeight` on FibWalk and Jumpable respectively.<br>Exploding a mushroom causes the message 'A mushroom exploded!' to the log. Mushrooms can only explode once.<br><br>The check whether the mushroom has exploded needs only be done at the end of the frame - if an object skips the mushroom by speeding past it within a single frame, no explosion will occur. This is to simplify your code.<br><br>Example for Multiplier([(0,2)], 1) on an object with Jumpable(2, 1, 1): If the object starts at (0, 0), at the beginning nothing happens. After a second has passed, Jumpable is activated the object jumps. Since the `triggerDistance` is 1, by the time the object reaches half the jump (the jump has a max height of 2, so half height is 1). It will be with `triggerDistance` of (0, 2) because \|(0, 1) - (0, 2)\| < 1. Now the 'A mushroom exploded!' message is logged and all other components get multiplied by 2. This means that the jumping becomes Jumpable(4, 1,1) (e.g. the object will get twice as high for the same effort) and thus the jump should now end at (0, 4) instead of (0, 2), albeit at much greater speed. The mushroom is now exploded and since it was the only one left, the component does nothing for the rest of the game.<br><br>Hint: For intra-object communication we suggest using events as discussed in the gameplay / gameplay for UE4 slides. |

- You are not allowed to use any timing information other than the delta time that we passing.
- We expect to receive all your source files and a project file (either .vcxproj or CMakeFileLists.txt)
    - All source files must be compilable against either VS2015, update 3 or VS2017
    - Building the project file ***MUST*** output a ***DYNAMIC LIBRARY*** (.DLL)
        - To create a dynamic library project just follow the VS project wizard. You can also check this tutorial.
- You are not allowed to use any third party libraries

- We expect to be receive a ***<fn>.hw1.zip*** archive containing the source and project files.
- Upload the archive to moodle (https://learn.fmi.uni-sofia.bg/course/view.php?id=2456#section-8)
- Cheaters are to be banished to the ethereal world, don't forget that.
- For any questions you are free to comment on this file as well as write to us directly however you see fit.

## Testing process

We'll build an executable linking against your dynamic library and we'll try to load our custom objects using your system, call Update several times and check whether the results comply with the expected output. Since we are dealing with a lot of floating point computations we don't expect your numbers to match exactly, we'll check them with some error margin.

## Complete example:

Input file (*test1.in*):

```
class Plumber
- Jumpable(10, 1, 5)
- FibonacciWalk(10, 9999999, 1)
- Renderable
```

Code:

```
auto world = CreateWorld();
world->ParseTypes("test1.in");
world->SpawnObject("Plumber", "Mario");
world->Update(1000.f);
world->SpawnObject("Plumber", "Luigi");
world->Update(1000.f);
world->Update(1e-4f);
world->Destroy();
```

Expected output located in *test1.out*; explanations given for each line in comments marked with #

```
--- Starting frame 0 ---
Mario is located at (1, 10) # Jumpable and FibWalk have already executed and since dt is exactly 1s, then Mario must be at the peak of his jump
--- Starting frame 1 ---
```

```
Mario is located at (2, 0) # Another second has passed, Mario is back to the ground, and fibwalk has moved 1 unit to the right
Luigi is located at (1, 10)  # This is the frame for Luigi, he's where Mario was at the last frame
--- Starting frame 2 ---
Mario is located at (4, 0) # Jumpable is now in delay for the next 5 seconds, the FibWalk moved it 2 units to the right
Luigi is located at (3, 9.99999) # Luigi has started falling but the delta time is close to 0 so falling back only moves Luigi an extremely small distance. FibWalk ignores time so it will work regardless.
```