# 1 Overview

The game executes the `update` method each cycle and these are the following things that happen:

```
/**
        * Update cycle:
        * - Detect Collisions (Collision System)
        * - Get input (Controller)
        * - Pass collision data to all systems (system_base)
        * -Pass input to all systems (system_base)
        * - Call update on each system
        * - Render (Render System)
**/
```

The update function returns a boolean and if it returns false it means the game must end and then the game proceeds to call the end_base() function on each system which is a template function and all systems can override a private end function to add behavior that can happen when the game ends.

# 2 Design

## 2.1 Entities

Each `EntityID` is just a unique 32 bit integer. The maximum number of entities that the game can have must be determined at compile time (Reason later).

```
using EntityID = std::uint32_t;
static const constexpr std::size_t MAX_ENTT = 5000;
```

The creation / destruction / Modification of these entities is done by a class called `EntityManager`. When a user creates an entity it can only see the `EntityID` returned and not the internals of the entity itself. The `Entity` itself is a nested `struct` inside the manager class

```
// EntityManager.h
struct Entity{
  bool isAlive{};
  ComponentSignature signature;
};
```

The reason for `ComponentSignature` being there will be explained later. What is important is user can only access entities by their *id*. The interface for creating / destroying entities is:

```
EntityID createEntity();
void destroyEntity(EntityID id);
```

## 2.2 Component

A `Component` is any piece of **pure** data. A *user* can attach any component to an entity and its behavior will change based on the data added. For example the some physics based components are:

```
// Components/default/default.h
struct Vec2d{ float x; float y; }
struct Position{
  Vec2d p;
  float z; // height
  float len;
  float width; };
struct Velocity{  Vec2d v; };
struct Acceleration{  Vec2d a; };
```

These are just pure pieces of *POD* data the user can attach to any entity and their behavior / movement will change based on that. Attaching / Detaching components to an entity is done with templates and to make it easier to *save* predefined data I've created a prefab template class.

```
template<typename T>  void attachComponent(EntityID id, T component)
template<typename T> void detachComponent(EntityID id)
// Examples:
GAME1.attachComponent<PlayerTag>(player {GAME1.createEntityFromPrefab(bullet), 0});
GAME1.attachComponent<AGE_COMPONENTS::Velocity>(er, {{0,0}});
GAME1.attachComponent<AGE_COMPONENTS::Solid_tag>(er, {});
```

### 2.2.1 Prefabs

This is a feature to make it easier to add stored entities with predefined data. So the user does not need to type out attach every time. For example the `bullet` prefab in the example above is defined as:

```
// Prefab.h
Prefab<AGE_COMPONENTS::Position, AGE_COMPONENTS::Drawable, AGE_COMPONENTS::Velocity, AGE_COMPONENTS::
    Acceleration, AGE_COMPONENTS::BoxCollider, AGE_COMPONENTS::Mass> bullet(
          "bullet",
          {{-1,-1},0, 0,0},
          {{{'o'}}},
          {{0,0}},
          {{0,0}},
          {1,1},
          {1.0f}
);
```

The list of components a `bullet` would need is huge and that is a reason defining prefab template saves

a lot of lines of code. Internally there are a lot of recursive template functions to make this work and this was one of the trickiest abstraction to get to work.

### 2.3 Component Array

All the data items for a component are stored in an array of size max_entt(5000) defined above. I've also made an optimization where all the created entities are guaranteed to be next to each other at the start of the array for locality of reference.

```
std::array<T, MAX_ENTT> entityData;
std::unordered_map<EntityID, size_t> entityMap;
std::unordered_map<size_t, EntityID> idxMap;
```

User can also define their own components like the `PlayerTag` from the example above but for the game to react to the data added the user must also define a `System` that would manage the data.

## 3 System

A `System` is the core logic and it handles mutating game state / movement of entities and the core game logic. To add a system to the game the interface is:

```
// AGEManager.h
template<typename T> std::shared_ptr<T> registerSystem()
```

Any system the user defines **must** have an *update* method and they can decide if they need an `end()` method which is called for each system at the end of the game. To enforce this on user defined systems the inheritance was the best strategy. Any system must be derived from `system_base` class. It also provides interfaces the user can access to get the current *Input*, *Collision* data for their system to react. A system can only run and modify entities that have a specified *signature*. That is they **must** have a specific list of components added to them. This information is represented in a compile time bitset since every time a user attaches a component they call a template function that is what makes it possible to make these bit sets at compile time.

For example for **PhysicsSystem** each physics entity, the entity must have a `Position`, `Velocity` and `Acceleration`. So inform this to the game manager the user must set the system *traits* after they've registered a system.

```
// AGEManager.h
template<typename Sys, typename First, typename Second,typename... Rest>
    void SetSystemTraits();
```

It is a varadic template function that sets the system's required signature based on the arguments passed in. For the **PhysicsSystem** example above the user would type:

```
mgr.safe_addComponent<AGE_COMPONENTS::Velocity>();
mgr.safe_addComponent<AGE_COMPONENTS::Acceleration>();
mgr.registerSystem<PhysicsSystem>();
mgr.SetSystemTraits<PhysicsSystem, AGE_COMPONENTS::Position, AGE_COMPONENTS::Velocity, AGE_COMPONENTS::
    Acceleration>();
```

Now any entity *atleast* with those components attached is passed through the PhysicsSystem each update. This is done by checking the bitwise AND on the systems required signature and an entities signature. To show the elegance of this approach the static physics system has an extremely simple update function:

```
// PhysicsSystem.h
 bool update() override{
        float dt = DeltaTime(); // provided by System_base
        for(const auto& et : Entities){ // contains all entities with required signature
            auto& pos = ComponentData.getComponentData<AGE_COMPONENTS::Position>(et);
            auto& vel = ComponentData.getComponentData<AGE_COMPONENTS::Velocity>(et);
            auto& acc = ComponentData.getComponentData<AGE_COMPONENTS::Acceleration>(et);
            auto v0 = vel.v;
            vel.v += acc.a*(dt);
            pos.p += (vel.v + v0)*0.5*dt;
        }
        return true;
    }
```

10 lines of code is enough to have linear movement in the system.

## 3.1 Special Systems / Components

The special systems out of all the user defined and provided physics systems in the library are the *RenderSystem* and the *CollisionSystem*. Since the order in which they execute matters. An overview of each update cycle in my game engine is:

```
/**
         * Update cycle:
         * - Detect Collisions
         * - Get input
         * - Pass collision data to all systems
         * -Pass input to all systems
         * - Call update on each system
         * - Render
**/
```

The Order of detecting collisions and rendering matters this is why these systems are highly coupled with the game manager itself and are defined very differently from all the other systems. Although they do follow the requirements of being systems they are just *special*. Rendering is special since it has access to the game drawing state and can mutate it. Collision system is special since the collision the collision system detects must be passed to other systems before calling their update function. In my engine the render system only renders components with a `Position` and a `Drawable` attached which is just a 2d vector of characters. The collision system only detects collisions on entities with a `BoxCollider` or a CircleCollider attached.

### 3.2 User Input

This is very simple in my engine each system can query is a key is pressed to the `Input` structure inside `system_base` and it returns true or false. A simple movement can be written as:

```
// Code from first game
auto& pos = ComponentData.getComponentData<AGE_COMPONENTS::Position>(et);
auto& v = ComponentData.getComponentData<AGE_COMPONENTS::Velocity>(et);
    if(Input.GetKey('w')){
     pos.p.y += 1; v.v.y += 2; }

// similar logic for all other keys ...
```

To detect key presses is done by the game manager and it passes that information to the Input structure each update cycle.

### 3.3 Drawing on the screen

At the end of each cycle a rvalue reference to the Game state vector is passed to the curses render class which draws out the characters on the screen. This can be replaced with any *view* that can render a vector of characters.

## 4 Sample Games

### 4.1 Compiling and Running

```
$ make
$ output/main <fps>
$ output/main2 <fps>
```

The executable files take an `<fps>` argument (default = 2).