Logic System

Procedural – Logic System

Logic System

- Separate the Game Engine from the Game logic and Game assets.
- In C++, ideally,
  - Have your game engine as a library (static ".lib" or dynamic ".dll").
  - Have your game as an application ".exe" that links to your engine library.
    - Has C++ Scripts
    - Has its own assets/resources.

Logic System

- Engine Side
  - LogicSystem
  - BehaviourFCT
  - BehaviourComponent
- Game Side
  - How to add a script?
  - How to add custom data component?
  - Using custom data component in a script

Logic System

- LogicSystem
  - In this sample code, we assume that your engine is using basic script's structure that only consumes 3 main functions:
    - » Init
    - » Update
    - » End
  - The functions will be wrapped in a BehaviourFCT class (or you can call it “Script” class)

Logic System

- LogicSystem
  - The Logic System holds 2 main containers
    - One container to hold the scripts (of type BehaviourFCT)
    - One container to hold the logic components

Logic System

- LogicSystem – .h

```
class LogicSystem : public ISystem //...
{
private:

    //various behaviours (contain scripts)
    std::vector<BehaviourFCT*> m_behaviours;

    //a behaviour-component contains the index into "m_behaviours"
    std::vector<BehaviourComponent*> m_behaviourComponents;

    //...

public:

    void Init();
    bool Update();
    void End();

    void AddBehaviour(BehaviourFCT * behaviour);

    //...

    ~Logic();//destructor deletes all the "BehaviourFCT*" objects
};
```

Logic System

- LogicSystem – .cpp

```
void LogicSystem::Init()
{
    //...
}

bool LogicSystem::Update()
{
    for (auto & iter : m_behaviourComponents)
    {
        m_behaviours[iter->GetBehaviourIndex()]->m_UpdateBehaviour(iter->GetOwner());
    }
}

void LogicSystem::End()
{
    //...
}
```

- Note that we are passing the owner (*the GameObject*) to the script-update

Logic System

- BehaviourFCT – .h

```
typedef void(*InitBehaviour)(GameObject*);
typedef void(*UpdateBehaviour)(GameObject*);
typedef void(*EndBehaviour)(GameObject*);

class BehaviourFCT //allows making behaviour objects
{
private:
    //3 global function pointers that define a script (on the game code side)
    //this is a containment method, trying to avoid inheritance and vtables.
        //Versus, making these 3 fcts as virtual=0 and forcing users to derive from
        //this class to make custom scripts, on the game code side
    InitBehaviour m_InitBehaviour;
    UpdateBehaviour m_UpdateBehaviour;
    EndBehaviour m_EndBehaviour;

    //...

public:
    BehaviourFCT(const InitBehaviour & Init, const UpdateBehaviour & Update, const EndBehaviour & End):
    m_InitBehaviour(Init),
    m_UpdateBehaviour(Update),
    m_EndBehaviour(End)
    {
        //attaching this to the LogicSystem
        MyLogicSystem.AddBehaviour(this);//MyLogicSystem can be unique instance of "LogicSystem"
    }
};
```

Logic System

- BehaviourComponent – .h

```
class BehaviourComponent : public IComponent
{
protected:

    //holds the behaviour index of the scripts container in the LogicSystem
    unsigned int m_behaviourIndex;

public:

    //...

    void SetBehaviourIndex(const unsigned int & behaviourIndex);
    inline unsigned int & GetBehaviourIndex() noexcept;
};
```

Logic System

- How to add a script, on the Game code side?

```
namespace NPC001_Script
{
    void Start(GameObject* gob)
    {
        //NPC001 Start script code - here
    }

    void Update(GameObject* gob)
    {
        //NPC001 Update script code - here
    }

    void End(GameObject* gob)
    {
        //NPC001 End script code - here
    }

    //This is one way to add a script to the engine, from the game code side
    //Not proper, but it works!
    BehaviourFCT * behaviourFCT = new BehaviourFCT(Start, Update, End);
}
```

Logic System

- How to add custom data component?
  - A script can be used by many game objects (entities)
  - What if we need customized data, to be used by the different game objects?
    - Answer:
      - Add a custom data component (i.e. *LogicData001*)
      - Each game object adds its own custom data component
      - Use/Access that custom data component in the script

Logic System

- How to add custom data component?
  - *LogicData001* component class example

```
class LogicData001 : public IComponent  //customized data - Game code side
{
public:
    LogicData001(const int & health) : m_health(health) {}

    //...

    LogicData001 * Clone() { return new LogicData001(*this); }

    inline int & GetHealth() noexcept { return m_health; }
    void SetHealth(const int & health) noexcept { m_health = health; }

    //...

private:

    //Data
    int m_health;
};
```

Logic System

- How to add custom data?
  - *LogicData001* component class registration
  - Add an “EngineCall” cpp file onto the Game code side

```
#include "LogicData001.h"

namespace MyAwesomeEngine
{
    //"Pre_GameEngine_Init" is a helper function that is declared in your
    //engine but not defined. It must be defined here, on the Game code side
    //"Pre_GameEngine_Init" is also called in your engine, before all your
    //systems initialize.
    void Pre_GameEngine_Init()
    {
        RegisterComponent(LogicData001);
    }
}
```

Logic System

- Using the custom data component in a script

```
#include "LogicData001.h"

namespace NPC001_Script
{
    void Start(GameObject* gob)
    {
        //NPC001 Start script code - here
    }

    void Update(GameObject* gob)
    {
        //NPC001 Update script code - here

        //e.g. using LogicData001 component
        LogicData001 * logicData001 = (LogicData001 *)(gob->GetComponent("LogicData001"));
        if(logicData001)
            logicData001->SetHealth(100);
    }

    void End(GameObject* gob)
    {
        //NPC001 End script code - here
    }


    //This is one way to add a script to the engine, from the game code side
    //Not proper, but it works!
    BehaviourFCT * behaviourFCT = new BehaviourFCT(Start, Update, End);
}
```

Logic System

- Reminder

```
GameEngine::Update()
{
    InputSystem.Update()
    LogicSystem.Update()
    PhysicsSystem.Update()
    //...
}
```

Logic System

- Extensions
  - Expand your scripts containers to
    - State Machines
    - Decision Trees
    - Behavior Trees
  - Visual Support
    - Have a Logic editor in your Level editor
  - Other scripting languages
    - Bind scripting languages like LUA, C#…
  - Event Driven – Logic System

Logic System

- Improvement
  - Improvement on the previously shown code:
    1. In “LogicSystem” class, “m_behaviours” can change to an “std::map”.
       - Therefore, in “BehaviourComponent” class “unsigned int m_behaviourIndex” becomes “std::string m_behaviourKey”.
    2. Class “BehaviourFCT” can be replaced with “Script” class that inherits from “IScript” class interface, and the 3 functions “Start”, “Update” and “End” become member functions, instead of pointing to 3 global functions.
       - This is changing from C-Style to C++ style.

Logic System Responsibility

- Updates component's data called by scripts
  - *player.rigidBody.ApplyForce(forceID);*
  - *ApplyPathFinding(AStar, npc1.transform.position, targetPos);*
  - *platform.rigidBody.ApplyForce(moveLeftForceID);*

  - *if(npc1.logicData.GetHealth() < 40)*
    - *npc1.sprite.SetAnimation(NPC_WEAK_ANIMATION);*

  - *AI::Build(AStar, waypointData, …); //at "Init"*

Logic System

- Additional helper functions used in scripts
  - Scene::GetGameObject(ID)
  - Scene::GetGameObjects(ID)
  - Scene::SpawnObject(prefabID)
  - Scene::DestroyObject(ID)
  - SceneManager::ChangeLevel(ID)
  - ...

Thank you!