Introduction to Engine Development with Component Based Design

Randy Gaul

Overview – Intro to Engine Development

- What is an engine
- Systems and game objects
- Components
- Engine
- Systems
- Messaging
- Serialization

What is an Engine?

- Collection of systems

```
class Engine
{
public:
  void RunGame( );
  void AddSystem( System *system );
  void ShutDown( );

private:
  std::vector<System *> m_systems;
  bool m_gameRunning;
}
```

Systems

- Perform operations
  - Often on game objects
- Example systems:
  - Graphics
  - Game Logic
  - Physics
  - Input
  - Object manager

```
class System
{
public:
  virtual void Update( float dt ) = 0;
  virtual void Init( void ) = 0;
  virtual void SendMessage( Message *msg ) = 0;

  virtual ~System( ) {}
};
```

Game Objects

- Collection of Components
- What is a component?
  - Contains functions
  - Contains data
  - More on this later
- Examples:
  - Sprite
  - AI/Behavior
  - Player controller
  - Physics collider

Game Object Example

```
class GameObject
{
public:
  void SendMessage( Message *msg );
  bool HasComponent( ComponentID id ) const;
  void AddComponent( ComponentID id, Component *c );
  void Update( float dt );
  Component *GetComponent( void );
  GameObjectHandle GetHandle( void ) const;
  bool Active( void ) const;

private:
  GameObjectHandle m_handle;
  std::vector<Component *> m_components;
  bool m_active;

  // Only to be used by the factory!
  GameObject( );
  ~GameObject( );
};
```

Game Object Details

- SendMessage
  - Forwards message to all components
  - Component can ignore or respond to each message type
- Active
  - Game objects and components contain active bool
  - False means will be deleted
  - ObjectFactory update should delete everything with m_active as false
- Handle
  - Unique identifier used to lookup a game object
  - Can be integer, use std::map to start with

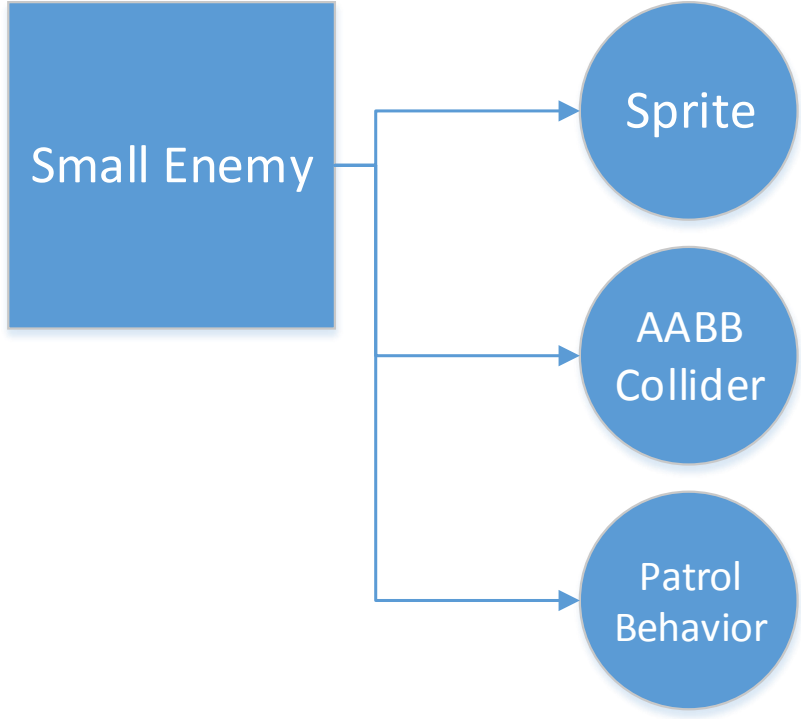
Game Object Details

- Private constructor/destructor
  - Only want ObjectManager system to create/destroy game objects

Components

- Components define game objects
  - Determine behavior, appearance, functionality
- Contains data
- Contains functions

Game Object Diagram



Base Component Example

```
class Component
{
public:
  virtual ~Component( );

  virtual void SendMessage( Message *message ) = 0;
  virtual void Update( float dt ) = 0;
  virtual void Init( void ) = 0;
  virtual void ShutDown( void ) = 0;
  bool Active( void ) const;

private:
  ComponentID m_id; // type of component
  bool m_active;
  GameObjectHandle m_owner;
};
```

Derived Component Example

- Physics Component Example

```
class Collider : public Component
{
public:
  // Implement base functions here
  void ApplyForce( const Vec2& force );
  void SetVelocity( const Vec2& vel );

private:
  Shape *shape;
  Vec2 m_forces;
  Vec2 m_velocity;
  Vec2 m_position;
};
```

Questions?

Systems You Will Need

- Object Manager/Factory
- Graphics
- Physics
- Game Logic
- Input/Windows Manager

Base System Class

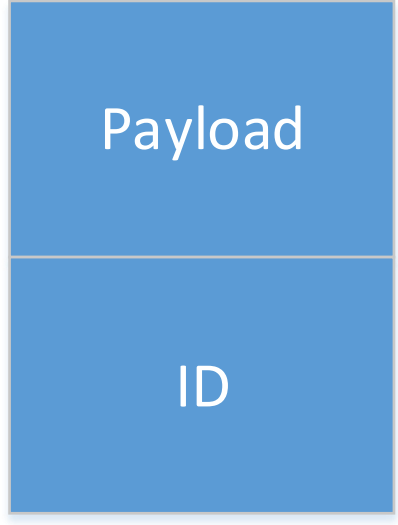
```
class System
{
public:
  virtual void Update( float dt ) = 0;
  virtual void Init( void ) = 0;
  virtual void SendMessage( Message *msg ) = 0;

  virtual ~System( ) {}
};
```

Systems You Might Want

- Audio Manager
- Resource Manager
- Memory Allocator
- Scripting
- UI Manager
- Anything else your heart desires

Messaging

- What is a message?
  - Payload
  - ID



Messaging

- Send information from one place to another
  - Virtual function call is type of messaging
- Can send info from any one place to any other

Messaging

- Send to Engine:
  - Usually just forward to all systems
- Send to system:
  - Each system handles messages uniquely
- Send to GameObject
  - Usually forwards to all components
- Send to Component:
  - Each component handles messages uniquely

The Need for Messaging

- Imagine this:
  - Systems, game objects and engine can receive messages
- Imagine an Enemy class
  - Has some functions
- How do you call a function?
  - #include Enemy.h

The Need for Messaging

- Imagine coding a battle sequence
  - Player fights monsters

```
Player::Battle( Enemy *enemy )
{
  enemy->SpitFireOnto( this );
}
```

The Need for Messaging

- Imagine coding a battle sequence
  - Player fights monsters

```
Player::Battle( Enemy *enemy )
{
  enemy->SpitFireOnto( this );
}
```

- Any problems?

The Need for Messaging

```
Player::Battle( Enemy *enemy )
{
  enemy->SpitFireOnto( this );
}

Player::Battle( SmallEnemy *enemy )
{
  enemy->SmallEnemyAttack( this );
}

Player::Battle( FatEnemy *enemy )
{
  enemy->Eat( this );
}
```

The Need for Messaging

```
#include "Enemy.h"
#include "SmallEnemy.h"
#include "FatEnemy.h"
```

The Need for Messaging

```
#include "Enemy.h"
#include "SmallEnemy.h"
#include "FatEnemy.h"
#include "DireEnemy.h"
#include "HealthPowerup.h"
#include "Lots of STUFF"
```

The Need for Messaging

```
#include "Enemy.h"
#include "SmallEnemy.h"
#include "FatEnemy.h"
#include "DireEnemy.h"
#include "HealthPowerup.h"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
#include "Lots of STUFF"
```

Messaging Purpose

- Lower file inclusions
- Generic way to send data from one place to another

Messaging Implementation

- Create base message class
  - Base just has ID
- Create many derived classes
  - Derived holds the payload
- Send base across SendMessage functions
  - Inside, use switch on ID to typecast

Processing a Message

```
// In some header
enum MessageID
{
  TakeDamage,
  FightBack
};
```

```
// In Enemy.cpp
Enemy::SendMessage( Message *message, int payload )
{
  switch(message->id)
  {
  case TakeDamage:
    int damage = payload;
    hp -= damage;
    break;
  case FightBack:
    HealthComponent *hp = (HealthComponent *)payload;
    hp->ApplyDamage( m_damage );
    this->PlayFightAnimation( );
    break;
  }
}
```

New Battle Sequence

```
Player::Battle( Enemy *enemy )
{
  // Star the player attack animation
  this->PlayAttackAnimation( );

  // Damage the enemy
  enemy->SendMessage( TakeDamage, m_damage );

  // Let the enemy fight back agains the player
  enemy->SendMessage( FightBack, this );
}
```

Payload

- Can use an integer
  - Typecast inside SendMessage to appropriate type
  - Can store 4 bytes
    - Able to store pointers as well
- Probably best to:
  - Store pointer to Message struct
  - Typecast inside SendMessage to derived types

Message Structs

```
// Base message
struct Message
{
  MessageID id;
};
```

```
// Random messages for game logic
struct DamageMessage : public Message
{
  int damage;
};

struct BumpIntoMessage : public Message
{
  GameObject *from;
  float collisionSpeed;
  float damage;
};
```

Serialization

- Write object to a file
- Read an object from a file
- Required for saving game state
- Quickly create new object types!!!
- Useful for level editors

Serialization

- Think in terms of components
- Write an object's components to file
- Read an object's components to file

```
Missile.txt
Position = { 12.5, 102.52 }
Health = 5
Sprite = Missile.png
GameLogic = MissileScript.txt
```

Serialization – Object Creation

- ObjectFactory – creates game objects
  - Can also create objects from a file

```
// Create a new game object
GameObject *o = ObjectFactory->CreateObject( "Missile" );
```

Serialization – Object Creation

```
GameObject *ObjectFactory::CreateObject( string fileName )
{
  // Open the corresponding file
  File file( fileName );

  // Construct the object from string
  GameObject *gameObject = m_creators[fileName]->Create( );

  // Deserialize the gameObject from the file data
  gameObject->Deserialize( file );

  return gameObject;
}
```

Serialization – Object Creation

- Constructing objects from string
  - Use dependency inversion
  - Use std::map to start with
  - http://www.randygaul.net/2012/08/23/game-object-factory-distributed-factory/

Serialization – Object Creation

- Deserialize the GameObject from string
  - Create components from file

```
Missile.txt
Position = { 12.5, 102.52 }
Health = 5
Sprite = Missile.png
GameLogic = MissileScript.txt
```

  - Position, Health, Sprite, GameLogic are all components
  - Assign values to each created component from file data

Serialization – Object Creation

```
void GameObject::Deserialize( File file )
{
  while(file.NotEmpty( ))
  {
    string componentName = file.GetNextWord( );
    Component *component =
      ObjectFactory->CreateComponent( file, componentName );
    this->AddComponent( component );
  }
}
```

Serialization – Object Creation

```
void GameObject::Deserialize( File file )
{
  while(file.NotEmpty( ))
  {
    string componentName = file.GetNextWord( );
    Component *component =
      ObjectFactory->CreateComponent( file, componentName );
    this->AddComponent( component );
  }
}
```

Serialization – Object Creation

```
Component *ObjectFactory::CreateComponent( File file, string name )
{
  Component *component = m_creators[name]->Create( );

  // Deserialize is a virtual function!
  component->Deserialize( file );

  return component;
}
```

Component Deserialize Example

- Keep it really simple
- Hard-code the data members to read in

```
void PhysicsComponent::Deserialize( file )
{
  x = file.ReadFloat( );
  y = file.ReadFloat( );
}
```

Component Serialize Example

- To-file is just as simple

```
void PhysicsComponent::Serialize( file )
{
  file.WriteFloat( x );
  file.WriteFloat( y );
}
```

Parting Advice

- Don’t worry about efficiency
- Ask upper classmen for help
- Read Chris Peter’s demo engine
- Keep it simple
- Email me: r.gaul@digipen
- http://www.randygaul.net/2012/08/23/game-object-factory-distributed-factory/