Multiplayer Game Programming

Lecture 6: Modularization and Serialization

ITP 484

Client / Server Modularization

- Keep Server Code off the Client
  - Performance Optimization
  - Disk Space Optimization
  - Security
- Keep Client Code off the Server
  - Memory Use Optimization
  - Graphics Driver independence
  - Platform independence
  - Prevents crashes introduced by client code

Example

```
class Kart
{
public:

	Kart();

	void HandleShellImpact( GreenShell inShell );

private:

	int 					mHealth;

	AdjustVelocityFromBounce( Kart inOtherKart );
	PlaySound( SoundClip* inSoundClip );
	PlayParticleEffect( ParticleEffect* inFX );

	SoundClip 			mShellHitSound ;
	ParticleEffect 		mShellHitParticleFX ;
};
```

Implementation

```
void Kart::HandleShellImpact(
      GreenShell inShell )
{
      //hits cause damage!
      mHealth -= 1;

      UpdateVelocityFromBounce( inShell );

      inShell->Destroy();

      PlaySound( mShellHitSound );
      PlayParticleEffect( mShellHitParticleFX );
}
```

New Classes

```
class KartCommon
{
public:
	KartCommon ();
	virtual void HandleShellImpact(
		GreenShell inShell );
private:
	AdjustVelocityFromBounce( Kart inOtherKart );
};

class KartServer : public KartCommon
{
public:
	KartServer();
	virtual void HandleShellImpact(
			GreenShell inShell ) override;
private:
	int			mHealth;
}
```

New Classes

```
class KartClient : public KartCommon
{
public:
      KartClient();
      virtual void HandleShellImpact(
                  GreenShell inShell ) override;
private:
      PlaySound( SoundClip* inSoundClip );
      PlayParticleEffect( ParticleEffect* inFX );

      SoundClip                  mShellHitSound ;
      ParticleEffectmShellHitParticleFX ;
}
```

Implementation

```
void KartServer::HandleShellImpact(
     GreenShell inShell )
{
     //hits cause damage!
     mHealth -= 1;
     UpdateVelocityFromBounce( inShell );
     inShell->Destroy();
}
```

Implementation

```
void KartClient::HandleShellImpact(
     GreenShell inShell )
{
     UpdateVelocityFromBounce( inShell );

     PlaySound( mShellHitSound );

     PlayParticleEffect(
               mShellHitParticleFX );
}
```

Review!

- Name 2 transport layer protocols
- Name 2 link layer protocols
- Briefly describe the three managers above the Stream Manager in the Tribes networking model
- Explain how to implement a guaranteed delivery status notification system such as the one the Tribes Connection Manager provides

General Object Sharing

- Persist an object across the network
  - At some point in time, state is synchronized across hosts
- Considerations
  - Who?
    - Which host dictates state?
  - What?
    - Which properties on which object, if not all
  - When
    - How frequently
  - How?
    - Implementation
  - Why?
    - Because blowing your friends up is fun

Tribes: Ghost Manager

- Who?
  - Server dictates state of all ghosted objects to clients
- What?
  - Whatever state is marked dirty for that client (simulation / dropped packets)
- When?
  - Fixed number of times per second, dictated by user on client and server, prioritized by simulation, when in scope
- How?
  - Coming soon…

Alternatives (non-exhaustive)

- Who?
  - Clients / Peers own subset of objects and replicate state to all others
  - Client / Peer own an input state object and replicate it (instead of special case)
  - Any host can request ownership of an object and replicate state if ownership granted

Alternatives (non-exhaustive)

- What?
  - All objects, all properties
  - All properties of any object marked with a single dirty flag
  - All volatile properties of dirty objects and then individually tracked state for less volatile properties

Alternatives (non-exhaustive)

- When
  - Frequency
  - Priority
  - Scope / Relevance
    - Heavily simulation dependent
  - Immediacy
    - Wait until next packet is ready
    - Wait until end of frame
    - Force packet and send immediately
      - hope you're not using TCP

How!

- State Tracking
- Replication
- Serialization

State Tracking

- Programmer sets dirty state bits
  - Efficient
  - Error prone
- Programmer sets dirty object bit
  - State bits manually calculated when packet ready
  - Less efficient
  - Less error prone
- Scripting Language sets everything automatically when properties change
  - Minimal efficiency!
  - Minimal errors!

Replication

- Sending the message to create an object from one host to another
  - Need to assign an ID to this object so it can receive state updates or destruction message

Serialization

- Converting an object (or graph of objects) to a linear form that can be record “serially” into a file or sent “serially” across a network.
- Reverse sometimes referred to as deserialization

Serialization Considerations

- How to represent which type of object is serialized
- How to efficiently (space-wise and time-wise) represent internal properties
- How to represent links between serialize objects and relink after serialization

Let's build a naïve object replication and serialization system

- Let's replicate cars from our MarioKart example

```
struct Vec3 { float mX, mY, mZ; }
struct Quat { float mX, mY, mZ, mW; }

class Kart
{
      char                    mName[ 128 ];
      uint32_t                mShellCount;
      Vec3                    mPosition;
      Vec3                    mVelocity;
      Quat                    mRotation;
}

sizeof( Kart ) = 172 bytes
```

Object Replication Data

Per object data block

```
    enum EObjRepInstruction
    {
        EORI_Create,
        EORI_Update,
        EORI_Destroy
    }

    enum EObjectType
    {
        EOT_Kart,
        EOT_GreenShell,
        EOT_QuestionMarkBlock,
        EOT_BananaPeel
    }
```

Object Replication Header

```
struct ObjRepHeader
{
    EObjRepInstruction                          mInstruction;
    EObjectType
    mType;
    uint32_t
                                                mID;
}
```

```
sizeof( ObjRepHeader ) +
sizeof( Kart ) = 180 bytes
```

How do we serialize data into a packet?

```
class MemoryOutputStream
{
public:
      uint8_t*          mBuffer;
            //memory allocated for buffer;
      uint32_t          mBufferCapacity;
            //maximum data that will fit in buffer
      uint32_t          mBufferHead;
            //current location to read/write

      bool  WriteData( const void* inData,
                       size_t inLength );

};
```

WriteData

```
bool MemoryOutputStream::WriteData ( const void* inData,
                                     size_t inLength )
{
        if( inLength + mBufferHead <= mBufferCapacity )
        {
                memcpy( mBuffer + mBufferHead, inData, inLength );
                mBufferHead += inLength ;

                return true;
        }
        else
        {
                return false;
        }
}
```

How to replicate a Kart

```
void WriteKartConstruction( MemoryOutputStream* inMOS,
                            const Kart* inKart )
{
        ObjRepHeader orh;
        orh.mInstruction = EORI_Create;
        orh.mType = EOT_Kart;
        orh.mID = GetIDForKart( inKart );

        inMOS->WriteData( &orh, sizeof( ObjRepHeader ) );
        inMOS->WriteData( inKart, sizeof( Kart ) );
}
```

How to receive replicated Kart

```
void ProcessPacket( MemoryInputStream* inMIS)
{
	ObjRepHeader orh;
	While(inMIS->ReadData( &orh, sizeof( ObjRepHeader ) )
	{
		ProcessHeader( &orh, inMIS);
	}
}
```

```
void ProcessHeader( const ObjRepHeader* inORH,
                    MemoryInputStream* inMIS )
{
   void* obj;
   switch( inORH.mInstruction )
   {
      case EORI_Create:
         obj = CreateObject( inORH.mType );
         gIDToObjectMap[ inORH.mID ] = obj;
         DeserializeIntoObject( obj, inORH.mType,
            inMIS );
         break;
      case EORI_Update:
         obj = gIDToObjectMap[ inORH.mID ];
         DeserializeIntoObject( obj,
            inORH.mType, inMIS );
         break;
   }
}
```

```
void* CreateObject(EObjectType inType)
{
	switch( inType )
	{
		case EOT_Kart:
			return new Kart();
			break;
		case EOT_GreenShell:
			return new GreenShell();
			break;
		…
	}
}
```

```
void DeserializeIntoObject( void* ioObj,
        EObjectType inType, MemoryInputStream* inMIS )
{
        switch( inType )
        {
                case EOT_Kart:
                        inMIS->ReadData( ioObj,
                                sizeof( Kart ) );
                        break;
                case EOT_GreenShell:
                        inMIS->ReadData( ioObj,
                                sizeof( GreenShell ) );
                        break;
                …
        }
}
```

Problems!

- Ugly, copy and pasted code!
- Grows bigger and bigger!
- Uses void* types
  - Compiler can't type check
- Object Replication Module depends on Gameplay code (Kart, GreenShell, etc.)
  - Limits code reusability
  - Decreases maintainability
  - Breaks unit testing
  - Makes your skin crawl

C++ to the rescue

- First step:
  - Make all replicated objects derive from base class

```
class RepObj
{
    …
};

class Kart : public RepObj
{
    …
};
```

Next, fix deserialization mess…

Teach each RepObj how to read and write itself…

```
class RepObj
{
	virtual bool Write( MemoryOuputStream* inMOS ) const = 0;
	virtual bool Read( MemoryInputStream* inMIS ) = 0;
};

class Kart : public RepObj
{
	virtual bool Write( MemoryOuputStream* inMOS ) const
override;
	virtual bool Read( MemoryInputStream* inMIS ) override;
};
```

```
bool Kart::Write(
        MemoryOutputStream* inMOS ) const
{
    return inMOS->WriteData( *this,
        sizeof( Kart ) );
}

bool Kart::Read( MemoryInputStream* inMIS )
{
    return inMIS->ReadData( *this,
        sizeof( Kart ) );
}
```

```
void ProcessHeader( const ObjRepHeader* inORH,
                    MemoryInputStream* inMIS )
{
      RepObj* obj;
      switch( inORH.mInstruction )
      {
            case EORI_Create:
                  obj = CreateObject( inORH.mType );
                  gIDToObjectMap[ inORH.mID ] = obj;
                  obj->Read( inPacketBuffer );
                  break;
            case EORI_Update:
                  obj = gIDToObjectMap[ inORH.mID ];
                  obj->Read( inPacketBuffer );
                  break;
      }
}
```

But CreateObject still a problem…

```
RepObj* CreateObject( EObjectType inType )
{
	switch( inType )
	{
		case EOT_Kart:
			return new Kart();
			break;
		case EOT_GreenShell:
			return new GreenShell();
			break;
		…
	}
}
```

How can we remove dependency on Kart, GreenShell, etc?

Hash Map!

- In c++11 we call it an unordered_map
- Map is often your best friend for eliminating unwanted dependencies
- Map from RepObj type to…
- function that creates RepObj of given type!

Function Pointer

- Lets you hold a function somewhere and call it later
- Works totally different than a method pointer, so don't try to shove a method into a function pointer
- Strongly typed, so must declare the exact type of the function
- In our case, takes nothing and returns a RepObj*

```
typedef RepObj* (*CreationFunc)()

unordered_map< EObjectType, CreationFunc >
gCreationFuncMap();

class Kart : public RepObj
{
public:
   static RepObj* Create() {return new Kart();}
   …
};

//somewhere in your initialization:
gCreationFuncMap[ EOT_Kart ] = Kart::Create;
```

Create Object is prettier now!

```
RepObj* CreateObject( EObjectType inType )
{
     CreationFunc cf = gCreationFuncMap[inType];
     return cf();
}
```