Multiplayer Game Programming

Lecture 7: Bandwidth Conservation

ITP 484

Bandwidth Conservation

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

Custom Write

```
bool Kart::Write(
	MemoryOutputStream* inMOS) const
{
	if(inMOS->GetFreeSpace() > GetRequiredKartBytes() )
	{
		inMOS->WriteData( mName, sizeof( mName) );
		inMOS->WriteData( &mShellCount,
			sizeof( mShellCount ) );
		inMOS->WriteData( &mPosition, sizeof( mPosition ) );
		inMOS->WriteData( &mVelocity, sizeof( mVelocity ) );
		inMOS->WriteData( &mRotation, sizeof( mRotation ) );
		return true;
	}
	return false;
}
```

Empty Array Truncation

Most of mName is probably empty…

```
inMOS->WriteData(mName, sizeof(mName));
```

Becomes

```
uint16_t length = strlen(mName);
inMOS->WriteData( &length, 2 );
inMOS->WriteData( mName, length );
```

Extend Packet Buffer

```
bool MemoryOutputStream::WriteString( const
std::string& inString )
{
	size_t length = inString.size();
	if( length < 65535 )
	{
		uint16_t l=static_cast< uint16_t >( length );
		WriteData( &l, 2 );
		return WriteData( &inString[ 0 ], length );
	}
	return false;
}
```

Extend Packet Buffer

```
bool MemoryInputStream::ReadString(
     std::string& outString )
{
     uint16_t length;
     if( ReadData( &length, 2 ) )
     {
          outString.resize( length );
          return ReadData(
               &outString[ 0 ], length );
     }
     return false;
}
```

"Compressed" Strings

```
class Kart
{
    std::string     mName;
    uint32_t        mShellCount;
    Vec3            mPosition;
    Vec3            mVelocity;
    Quat            mRotation;
};
```

New Write

```
bool Kart::Write( MemoryOuptutStream* inMOS) const
{
	if(InMOS ->GetFreeSpace() > GetRequiredKartBytes() )
	{
		InMOS->WriteString( mName );
		InMOS->WriteData( &mShellCount,
			sizeof( mShellCount ) );
		inMOS->WriteData( &mPosition, sizeof( mPosition ) );
		inMOS->WriteData( &mVelocity, sizeof( mVelocity ) );
		inMOS->WriteData( &mRotation, sizeof( mRotation ) );
		return true;
	}
	return false;
}
```

More bits than we need!

```
class Kart
{
    std::string         mName;
    uint32_t            mShellCount;
        //game allows max of 3 shells
    Vec3                    mPosition;
    Vec3                    mVelocity;
    Quat
    mRotation;
}
```

Bit Stream Support

```
class MemoryOutputBitStream
{
public:
	uint8_t*			mBuffer;
				//memory allocated for buffer
	uint32_t			mBufferBitCapacity;
				//maximum data that will fit in buffer
	uint32_t			mBufferBitHead;
				//current bit location to read/write

	bool	WriteBits( uint8_t inValue, size_t inBitCount);
	bool	ReadBits( uint8_t& outValue, size_t inBitCount);
	bool	WriteBits( const void* inData, size_t inBitCount);
	bool	ReadBits( void* outData, size_t inBitCount );

};
```

```
bool MemoryOutputBitStream::WriteBits( uint8_t inValue, size_t
inBitCount)
{
    if( mBufferBitHead + inBitCount <= mBufferBitCapacity )
    {
        uint32_t byteOffset = mBufferBitHead >> 3;
        uint32_t bitOffset = mBufferBitHead & 0x7;
        mBuffer[ byteOffset ] |= inValue << bitOffset;

        uint32_t bitsFreeThisByte = 8 - bitOffset;
        if(bitsFreeThisByte < inBitCount)
        {
            //we need another byte…
            //note we shift left by bitOffset
            //and then right by 8
            mBuffer[ byteOffset + 1 ] = inValue >> bitsFreeThisByte;
        }

        mBufferBitHead  += inBitCount;

        return true;
    }

    return false;
}
```

Multiple Bytes

```
bool MemoryOutputBitStream::WriteBits( const void* inData, size_t
inBitCount)
{
        if( mBufferBitHead + inBitCount <= mBufferBitCapacity )
        {
                uint8_t* srcByte = reinterpret_cast< uint8_t* >( inData );
                //write all the bytes
                while( inBitCount > 8 )
                {
                        WriteBits( *srcByte, 8 );
                        ++srcByte;
                        inBitCount -= 8;
                }
                //write anything left
                if( inBitCount > 0 )
                {
                        WriteBits( *srcByte, inBitCount );
                }
                return true;
        }
        return false;
}
```

New Write

```
bool Kart::Write( MemoryOutputBitStream* inMOBS ) const
{
	if(inMOBS->GetFreeSpace() > GetRequiredKartBytes() )
	{
		inMOBS->WriteString( mName );
		inMOBS->WriteBits( &mShellCount, 2 );
		inMOBS->WriteBytes( &mPosition, sizeof( mPosition ) );
		inMOBS->WriteBytes( &mVelocity, sizeof( mVelocity ) );
		inMOBS->WriteBytes( &mRotation, sizeof( mRotation ) );
		return true;
	}
	return false;
}
```

Only send necessary properties!

```
Enum KartStateBits
{
    KSB_Name                = 1 << 0,
    KSB_ShellCount          = 1 << 1,
    KSB_Pose                = 1 << 2,
    KSB_Velocity            = 1 << 3
}
```

```
bool Kart::Write( MemoryOutputBitStream* inMOBS, uint32_t inStateBits )
const
{
	if(inMOBS->GetFreeSpace() > GetRequiredKartBytes() )
	{
		inMOBS->WriteBits( &inStateBits, 4 );
		if(inStateBits & KSB_Name  )
		{
			inMOBS->WriteString( mName );
		}
		if(inStateBits & KSB_ShellCount )
		{
			inMOBS->WriteBits( &mShellCount, 2 );
		}
		if( inStateBits & KSB_Pose )
		{
			inMOBS->WriteBytes( &mPosition, sizeof( mPosition ) );
			inMOBS->WriteBytes( &mRotation, sizeof( mRotation ) );
		}
		if( inStateBits & KSB_Velocity )
		{
			inMOBS->WriteBytes( &mVelocity, sizeof( mVelocity );
		}
		return true;
	}
	return false;
}
```

```
bool Kart::Read( MemoryOutputBitStream* inMOBS, uint32_t&
outStateBits )
{
   inMOBS->ReadBits( &outStateBits, 4 );
   if(outStateBits & KSB_Name  )
   {
      inMOBS->ReadString( mName );
   }
   if(outStateBits & KSB_ShellCount )
   {
      inMOBS->ReadBits( &mShellCount, 2 );
   }
   if(outStateBits & KSB_Pose )
   {
      inMOBS->ReadBytes( &mPosition, sizeof( mPosition ) );
      inMOBS->ReadBytes( &mRotation, sizeof( mRotation ) );
   }
   if(outStateBits & KSB_Velocity )
   {
      inMOBS->ReadBytes( &mVelocity, sizeof( mVelocity );
   }

   return true;
}
```

Only send necessary precision!

- Position units in meters
- World 1k by 1k
- Elevation changes max of 20m
- Client "correctness" to within .1 m
  - Interpolation and lag obviate higher precision
- Biggest number 10000 times smallest
- Don't need 32 bit float to represent this
  - Just need 14 bits, since 2^14 is 16384

Only send necessary precision!

- Quantization
  - Round numbers to nearest relevant value
- Fixed Point
  - Convert floats to fixed point integers
  - Use knowledge of max bounds and precision

```
bool MemoryOutputBitStream::WriteVec3(
                                        const Vec3& inVec )
{
   if( inBuffer->GetFreeBitCount() < 36 )
   {
      uint32_t fixedX =
         static_cast< uint16_t >( inVec.mX * 10.f );
      WriteBits( &fixedX, 14 );
      uint32_t fixedY =
         static_cast< uint16_t >( inVec.mY * 10.f );
      WriteBits( &fixedY, 14 );
      uint32_t fixedZ =
         static_cast< uint8_t >( inVec.mZ * 10.f );
      WriteBits( &fixedZ, 8);
      return true;
   }
   return false;
}
```

```
bool MemoryInputBitStream::ReadVec3(
                                const Vec3& inVec )
{
    uint32_t fixedX = 0;
    ReadBits( &fixedX, 14 );
    inVec.mX =
       static_cast<float>(fixedX) / 10.f;
    uint32_t fixedY = 0;
    ReadBits( &fixedY, 14 );
    inVec.mY =
       static_cast<float>(fixedY) / 10.f;
    uint32_t fixedZ = 0;
    ReadBits( &fixedZ, 8 );
    inVec.mZ =
       static_cast<float>(fixedZ) / 10.f;
}
```

Use single bit for default value!

- Cars are usually on the road
- Let's have z default to zero

```
bool MemoryOutputBitStream::WriteVec3(
                                const Vec3& inVec )
{
   if( inBuffer->GetFreeBitCount() < 37 )
   {
      uint32_t fixedX =
         static_cast< uint16_t >( inVec.mX * 10.f );
      WriteBits( &fixedX, 14 );
      uint32_t fixedY =
         static_cast< uint16_t >( inVec.mY * 10.f );
      WriteBits( &fixedY, 14 );
      bool shouldWriteZ = ( inVec.mZ > 0.f );
      WriteBits( &shouldWriteZ, 1 );
      if( shouldWriteZ )
      {
         uint32_t fixedZ =
            static_cast< uint8_t >( inVec.mZ * 10.f );
         WriteBits( &fixedY, 8);
      }
      return true;
   }
   return false;
}
```

```
bool PacketBuffer::ReadVec3( const Vec3& inVec )
{
      uint32_t fixedX = 0;
      ReadBits( &fixedX, 14 );
      inVec.mX =
         static_cast<float>(fixedX) / 10.f;
      uint32_t fixedY = 0;
      ReadBits( &fixedY, 14 );
      inVec.mY =
         static_cast<float>(fixedY) / 10.f;
      bool shouldReadZ = false
      ReadBits( &shouldReadZ, 1 );
      if( shouldReadZ )
      {
         uint32_t fixedZ = 0;
         ReadBits( &fixedZ, 8 );
         inVec.mZ =
            static_cast<float>(fixedZ) / 10.f;
      }
      else
      {
         inVec.mZ = 0.f;
      }
}
```

Quaternions too!

- Magnitude = 1
  - infer 4^{th} component from first 3
- Each component <= 1
  - 16 bit fixed point, maybe fewer
- Shrinks from 32 * 4 bits to 16 * 3 bits!