CH11_008

ANSWER

MYSECONDSCENEOBJECT

1 mySecondSceneObject

Creating Initial Class

The very first step to creating the new class is the creation of a new set of files and their insertion into the "Torque Demo" build files.

To do this, I simply copied "myFirstSceneObject.h" and "myFirstSceneObject.cc" into a new directory under the "engine/" directory.

Then, I renamed the files to "mySecondSceneObject.h" and "mySecondSceneObject.cc".

Next, I added the files to my build files and opened them in an editor.

Last, I did a global search-and-replace on each file, changing "myFirstSceneObject" to "mySecondSceneObject".

Test Compile

Before writing any new code, I did a test compile and test run to be sure that the class was working. (Remember, there is a mission associated with this exercise, and it tries to place one instance of mySecondSceneObject in the mission.)

When I ran the mission, I saw a single pyramid rendered in the mission.

Render Control Variable

The very first new code I decided to write was the code for the (render) control variable.

I knew I needed a variable that could hold integer values. I also knew I would have to expose this variable to the console and to transmit it over a network. So, I chose an S32 value, which can accomplish all these tasks.

```
class mySecondSceneObject : public SceneObject
{
public:
   mySecondSceneObject();
   ~mySecondSceneObject();

   S32 mDrawControl;
```

MYSECONDSCENEOBJECT

Also, according to the class parameters, this variable will have to hold the values 0x1, 0x2, or 0x3. So I decided to initialize it to 0x3 to start.

```
mySecondSceneObject::mySecondSceneObject()
{
   mDrawControl = (0x1 | 0x2);

   mTypeMask |= StaticObjectType | StaticTSObjectType |
                StaticRenderedObjectType;
   mNetFlags.set(Ghostable);
}
```

Rendering Cube and Pyramid

Now that I had a control variable, I switched my focus to the render code.

To satisfy the requirements of this exercise, I merely stole the render code from myGameBase/ myGameBaseData and plugged it into my new class.

First, the declarations.

```
class mySecondSceneObject : public SceneObject
{
private:
   typedef SceneObject      Parent;

   void DrawGLPyramid(void);
   void DrawGLCube(void);

public:
   S32 mDrawControl;
```

Second, the render routines.

```
void mySecondSceneObject::DrawGLPyramid(void)
{
   //  ... Same code as in myGameBase
}
```

MYSECONDSCENEOBJECT

```
void mySecondSceneObject::DrawGLCube(void)
{
   //  ... Same code as in myGameBase
}
```

Third, a slight modification to the renderObject() method to render these two objects and to use a control check.

```
void mySecondSceneObject::renderObject(SceneState* state,
                                       SceneRenderImage*)
{
   // ... Standard init code

   // New render code and logic
   if( 0x1 & mDrawControl ) DrawGLPyramid();

   // Reset the render matrix before drawing again.
   glPopMatrix(); // Pop stored ModelView Matrix
   glPushMatrix(); // Push current ModelView Matrix
   dglMultMatrix(&getTransform());

   if( 0x2 & mDrawControl ) DrawGLCube();

   // ... Standard cleanup code
}
```

Now (when I re-compiled and rendered), I had two shapes rendering , but I was still missing the ability to control which one is rendered.

MYSECONDSCENEOBJECT

Exposing as an EnumTable

To add the ability to modify the control variable, I needed to expose the variable as field. Additionally, the requirements stated that this field should use the enum feature supplied by addField().

So, I first created the enum declaration.

```
class mySecondSceneObject : public SceneObject
{
private:
   typedef SceneObject      Parent;

   void DrawGLPyramid(void);
   void DrawGLCube(void);

public:
   mySecondSceneObject();
   ~mySecondSceneObject();

   enum DrawTypes {
      PyramidOnly    = 0x1,
      CubeOnly       = 0x2,
      PyramidAndCube = 0x3
   };

   S32 mDrawControl;
```

Then, I defined an EnumTable in the source file.

```
static EnumTable::Enums myRenderEnums[] =
{
   { mySecondSceneObject::PyramidOnly, "PyramidOnly" },
   { mySecondSceneObject::CubeOnly, "CubeOnly" },
   { mySecondSceneObject::PyramidAndCube, "PyramidAndCube" }
};

static EnumTable gMyRenderEnumTable( 3 , myRenderEnums );
```

MYSECONDSCENEOBJECT

Last, I exposed the variable and a persistent field.

```
void mySecondSceneObject::initPersistFields()
{
   Parent::initPersistFields();

   addField("drawControl", TypeEnum, Offset( mDrawControl,
            mySecondSceneObject), 1, &gMyRenderEnumTable,
            "Render Control");
}
```

Send and Receive Only Bits Required In Pack/Unpack

At this point, the only thing that was left to do was to update the packUpdate() and unpackUpdate() methods to ensure that ghosts are updated when the persistent-field/variable gets updated.

However, there was one hitch; the requirements stated that we can only send the minimum number of bits to represent the control value.

A close look at the control variable values will make it clear that only the first two bits are ever used.

- 0x1 - Bit 0 set.
- 0x2 - Bit 1 set.
- 0x3 - Bits 0 and 1 set.

With this in mind, I took a look at the "Stream References" in appendix B and found the following two stream methods.

1. void writeSignedInt()(S32 value, S32 bitCount) - Writes an S32 and specifies how many bits to send in an update.
2. S32 readSignedInt()(S32 bitCount) - Reads an S32 value from the stream, but only extracts the specified number of bits to represent the value.

MYSECONDSCENEOBJECT

Therefore, I wrote the two required methods like this.

```
U32 mySecondSceneObject::packUpdate(NetConnection * con, U32 mask,
                                    BitStream * stream)
{
   U32 retMask = Parent::packUpdate(con, mask, stream);

   stream->writeAffineTransform(mObjToWorld);
   stream->writeSignedInt(mDrawControl,2);

   return(retMask);
}

void mySecondSceneObject::unpackUpdate(NetConnection * con,
                                       BitStream * stream)
{
   Parent::unpackUpdate(con, stream);

   MatrixF      ObjectMatrix;
   stream->readAffineTransform(&ObjectMatrix);
   setTransform(ObjectMatrix);

   mDrawControl = stream->readSignedInt(2);
}
```

With this last change in place. I recompiled, re-ran the exercise mission, and hey... something went wrong! I could control rendering, but the cube wouldn't render...hmmm? Ah,yes.

I forgot that the stream methods were sending a signed value. This meant that I needed one additional bit to represent the sign (positive or negative).

So, I adjusted the bit count from 2 to 3, and viola! It worked.

```
   stream->writeSignedInt(mDrawControl,3);
// ...
   mDrawControl = stream->readSignedInt(3);
```