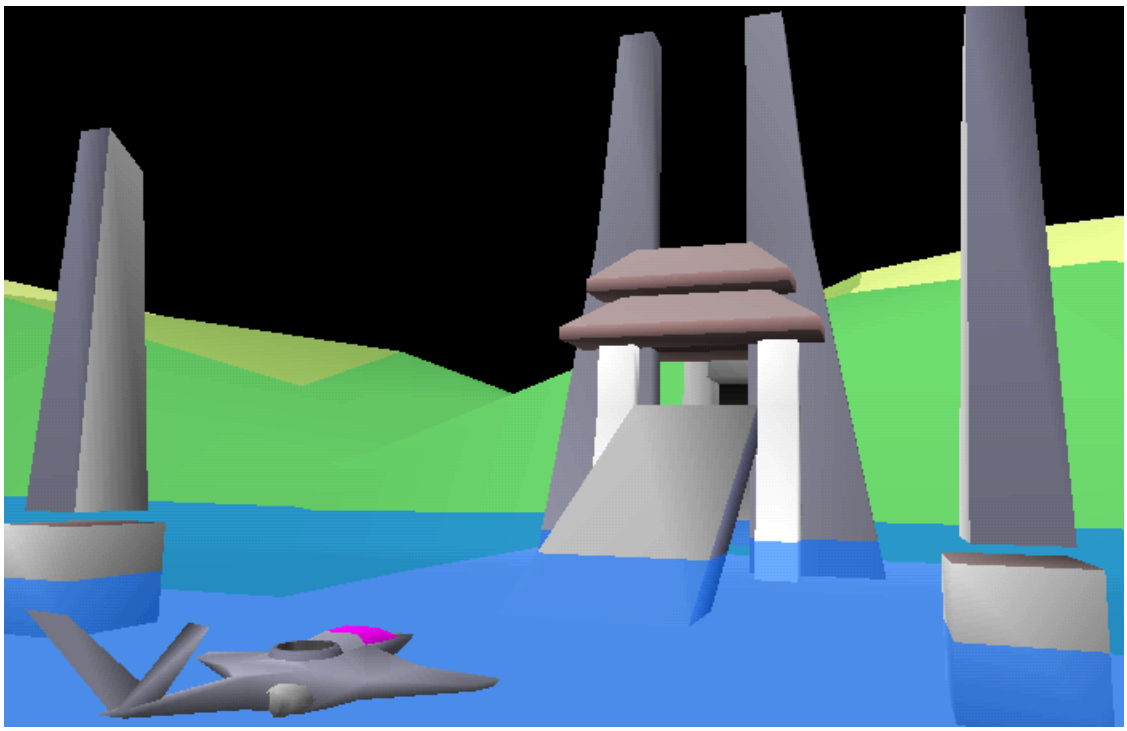
Building Virtual Reality Applications With

Amber V1.3



AMBER TUTORIALS

LESSON 1 - A BASIC AMBER APPLICATION

Part 1 - Creating the Single Document Application.

Part 2 - Adding Amber Source Code

LESSON 2 - ADDING A 6DOF MOUSE

Part 1 - Adding An Amber Mouse Sensor

Part 2 - Changing Universe Attributes

Part 3 - Attaching The Mouse To Objects

LESSON 3 - ADDING BEHAVIORS

Part 1 - Adding Styles

Part 2 - Adding Actions

Part 3 - Adding Laws

Amber Tutorials

This guide is designed to teach an application programmer the basics of creating a virtual reality application using Amber. The guide follows a series of tutorials that start with a simple application and incrementally adds capabilities. The user is advised to follow each tutorial as it will demonstrate some of the key features of Amber.

This tutorial assumes that the user is familiar with Microsoft VisualC++. Tutorials for other compilers will be available from the DIVE Laboratories WEB site.

The tutorials can be found off of the main Amber install directory in the "tutorial" directory. Each lesson chapter has a corresponding lesson directory. For example, the complete lesson 1 source code can be found in the "lesson1" directory. In these directories are the actual source code and make files of the completed lesson. If the user does not wish to create a lesson from scratch, they can simply load the make file into Microsoft VisualC++, and find the changes in the code that are commented as "AMBER SPECIFIC".

Lesson 1 - A Basic Amber Application

Lesson 1 will create a simple application that creates an Amber instance, opens one universe and one channel, and loads a single simple model. All new Amber source line examples are in bold font. Repeated Amber source lines will only have the new source lines in Bold font.

Lesson 1 begins with the creation of a single document application under Microsoft VisualC++. To do this, use the Microsoft VisualC++ interface to create a new project. From the MFC AppWizard, you will be prompted by a series of steps.

Part 1 - Creating the Single Document Application.

Step 1. Select single document and then hit the NEXT button.
Step 2. Hit the NEXT button.
Step 3. Hit the NEXT button.
Step 4. Hit the ADVANCED button. In the "DOC Type Name" type "VRML". In the "File Extension" type "wrl" and return. Then, hit the NEXT button.
Step 5. Select "Use MFC in a static library" and hit the NEXT button.
Step 6. Hit the FINISH button. Hit OK when prompted.

You have now created a simple single document view application. You should have the following files in your project.

```
lessodoc.cpp    // Document class
lesson1.cpp     // Application class
lesson1.rc      // Application resources
lessovw.cpp     // Document Viewer class
mainfrm.cpp     // Standard main frame
readme.txt      // Readme
stdafx.cpp      // Standard file for precompiling headers
```

From the project window, select the "Win32 Release" option.

In order for this application to function as a proper Amber application, you must now override some of MFC's standard functions.

First, in the "lesson1.cpp" file, we need override one function.

Step 1. Select the AppWizard button. In the "Message Maps" section, select **CLesson1App** from the "Class Name" window. In the "Messages" scroll window, select the **ExitInstance** function and hit the ADD FUNCTION button.
Step 2. In the "Message Maps" section, select **CLesson1Doc** from the "Class Name" window. In the "Messages" scroll window, select the **OnOpenDocument** function and hit the ADD FUNCTION button. Then select the **WM_SIZE** function and hit the ADD FUNCTION button.
Step 3. In the "Message Maps" section, select **CLesson1View** from the "Class Name" window. In the "Messages" scroll window, select the **OnInitialUpdate** function and hit the ADD FUNCTION button. Then select the **WM_SIZE** function and hit the ADD FUNCTION button. Finally select the **PreCreateWindow** function and hit the ADD FUNCTION button.
Step 4. Hit the OK button and exit the App Wizard.

Before we continue, we must add the Amber library, and the Amber include paths to our project. From the "Project" menu, select "Settings". Select the "C++" tab of the window and then scroll down the

"Category" window and select "Preprocessor". In the "Preprocessor Definitions" window, add the following define:

AMBER_FLOAT_LIB

Then, in the "Additional Include Directories" window, add the path to the Amber include files. For the tutorials it will be "..\..\include". Then hit the OK button to return to the main project window.

Now you must add the proper libraries to link in for Amber and OpenGL. First from the "Project" menu, select "New Group" and name it "LIB". This will be the holder for all of our new libraries. With the right mouse button, select this new folder in the project window. When the popup window comes up, scroll down and select files. This brings up the "Project Files" window that you will need to add the necessary libraries to you project. In the "List Files Of Type" window, select "Library Files". From Amber's "LIB" directory, add the following file to the project:

AMBERF.LIB

Verify that you have selected the "Win32 Release" option from the project window. If you wish to use the "Win32 Debug" option, substitute the AMBERF.LIB file with AMBERFD.LIB file. This is the debug version of the Amber library.

From the VisualC++ "LIB" directory, add the following files:

GLU32.LIB
GLAUX.LIB
OPENGL32.LIB

Hit the "CLOSE" button to return to the main project window.

Part 2 - Adding Amber Source Code

We will now begin to actually add source lines specific to Amber. First, edit the "lesson1.cpp" file so that we can create our Amber instance as well as our main universe.

At the top of the file, add the following include file entry:

```
#include "stdafx.h"
#include "lesson1.h"

#include "mainfrm.h"
#include "lessodoc.h"
#include "lessovw.h"
#include "amber.hpp"
```

Before any other Amber code is executed, an Amber instance must be created. The main Amber instance object *amber* is defined in the "amber.hpp" file. To use it, add the following lines in the **InitInstance** function of the **CLesson1App** class:

```
universeClass *univ;
BOOL CLesson1App::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the
size
```

```
    //  of your final executable, you should remove from the
following
    //  the specific initialization routines you do not need.

    Enable3dControls();

    LoadStdProfileSettings();  // Load standard INI file options
                                    // (including MRU)

    // Amber instance initialiation
    //
    amber = new amberClass();

    // Main universe
    //
    univ = new universeClass();
```

You now have created the main Amber instance and a single universe with which you will add an object. In order for the application to properly clean up after exiting, you must add the following line to the **ExitInstance** function.

```
int CLesson1App::ExitInstance()
{
    // TODO: Add your specialized code here and/or call the base
class

    // Delete Amber instance
    //
    delete amber;
```

This call will delete the Amber instance as well as all vertices, polygons, geometries, channels, and universes. This is because all of these objects are under memory control of the Amber instance.

Now edit the "lessovw.h" file. This is the class definition file for the **CLesson1View** class. Here we will be adding some public data to the class definition. First, in the include section of the file, add the following line:

```
// lessovw.h : interface of the CLesson1View class
//
/////////////////////////////////////////////////////////////////////
////////

#include "channel.hpp"
```

Then, in the public section of the class, add the following line:

```
class CLesson1View : public CView
{
protected: // create from serialization only
    CLesson1View();
    DECLARE_DYNCREATE(CLesson1View)

// Attributes
public:
    CLesson1Doc* GetDocument();
```

```
    // Our Amber channel.
    channelClass *ch;
```

This will allow us to use the document view as a channel, or viewport, into the universe. Amber channels prepare a document view to be used as an OpenGL rendering window.

Now edit the "lessovw.cpp" file.

In the header section of the file, add the following line:

```
// lessovw.cpp : implementation of the Clesson1View class
//

#include "stdafx.h"
#include "lesson2.h"

#include "lessodoc.h"
#include "lessovw.h"

#include "amber.hpp"
```

In the class constructor, add the following lines:

```
CLesson1View::CLesson1View()
{
    // TODO: add construction code here

    // Initialize Amber channel
    ch = NULL;

}
```

This will initialize our channel parameter to NULL. This is necessary due to an "IF" statement that we will be using shortly. In the **OnInitialUpdate** function, add the following lines:

```
void CLesson1View::OnInitialUpdate()
{
    // TODO: Add your specialized code here and/or call the base
class

    // Retrieve the HWND pointer to this window
    //
    HWND hwnd = GetSafeHwnd();

    // Create the channel if it has not already been created
    //
    if (ch == NULL) {
          ch = new channelClass((void *)hwnd);
    }

    CView::OnInitialUpdate();
}
```

An important feature of Amber is that it will automatically adjust the viewing attributes whenever the user changes the window size. In order for this to be accomplished, add the following lines to the **OnSize** function:

```
void CLesson1View::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here

    // Adjust Amber channel when window size is changed.  Return if
    // channel has not been created or this window is not visible.
    //
    if (!IsWindowVisible() || ch==NULL) return;
    ch->resetPerspective();
}
```

In order for a window to properly be used as an OpenGL context, it must have certain attributes. These attributes are set by adding the following lines to the **PreCreateWindow** function:

```
BOOL CLesson1View::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Add your specialized code here and/or call the base
class

    // Set the style of the window to clip children and siblings.
    // This is a requirement of OpenGL windows.
    //
    cs.style = cs.style | WS_CLIPSIBLINGS | WS_CLIPCHILDREN;

    return CView::PreCreateWindow(cs);
}
```

You can now test this code by compiling and executing it. Upon execution, you should see a standard single document application frame with a window that has a black background. When the channel is open, our position is at the center of the world (0.0,0.0,0.0) looking down the -Z axis. We will now add an object to the world.

At the top of the "lessovw.cpp" file, add the following function:

```
void myInit()
{
    cubeClass *cube;
    V3 pos;

    // Create a unit cube
    //
    cube = new cubeClass(1,1,1);

    // Set the cube's position 10 units
    // away from the viewpoint.
    //
    V3_set(pos, 0.0, 0.0, -10.0);
    cube->setPosition(pos);

    // Rotate the cube so that we can get
```

```
    // some perspective without moving
    //
    cube->rotate(Y, 45*DEG_TO_RAD);
    cube->rotate(X, 20*DEG_TO_RAD);
}
```

This function will create a unit cube that is 10 units away from the viewpoint. It will also rotate the cube 45 degrees on its Y axis and 20 degrees on its X axis.

Then, in the **OnInitialUpdate** function, add the following line to the code:

```
void CLesson1View::OnInitialUpdate()
{
    // TODO: Add your specialized code here and/or call the base
class

    // Retrieve the HWND pointer to this window
    //
    HWND hwnd = GetSafeHwnd();

    // Create the channel if it has not already been created
    //
    if (ch != NULL) {
          ch = new channelClass((void *)hwnd);
          myInit();
    }

    CView::OnInitialUpdate();
}
```

Recompile and execute the program. You have now successfully create your first Amber application. You have demonstrated the ability to create an Amber instance, create a universe, create a channel for rendering, and create an object for viewing.

Please proceed to lesson 2.

Lesson 2 - Adding a 6DOF Mouse

Lesson 2 expands upon the application created in Lesson 1. In this lesson, we will add a mouse sensor to the application so that the channel viewpoint can be moved about the screen freely. This sensor has six degrees-of-freedom (6DOF). This means that the user can translate along any axis as well as rotate around any axis. This allows the user to view a single model, or the entire universe, from any point and orientation they chose.

Part 1 - Adding An Amber Mouse Sensor

Edit the "lessovw.h" file. This is the class definition file for the **CLesson2View** class. Here we will be adding some public data to the class definition. First, in the include section of the file, add the following line:

```
// lessovw.h : interface of the CLesson1View class
//
/////////////////////////////////////////////////////////////////////////
////////

#include "channel.hpp"
#include "mouse.hpp"
```

Then, in the public section of the class, add the following line:

```
class CLesson2View : public CView
{
protected: // create from serialization only
    CLesson2View();
    DECLARE_DYNCREATE(CLesson2View)

// Attributes
public:
    CLesson2Doc* GetDocument();

    // Our Amber channel.
    channelClass *ch;

    // Our Mouse Sensor.
    mouseClass *mouse;
```

This will provides the channel view with a mouse sensor. Each view should always get its own mouse sensor. You can use one mouse sensor for all open channels, but management of the mouse events becomes a bit more difficult.

Edit the "lessovw.cpp" file and In the class constructor, add the following lines:

```
CLesson2View::CLesson2View()
{
    // TODO: add construction code here

    // Initialize Amber channel
    ch = NULL;
```

```
    // Initialize Amber mouse
    mouse = NULL;
}
```

This will initialize our mouse parameter to NULL. This is necessary due to an "IF" statement that we will be using shortly. In the **OnInitialUpdate** function, add the following lines:

```
void CLesson2View::OnInitialUpdate()
{
    // TODO: Add your specialized code here and/or call the base
class

    // Retrieve the HWND pointer to this window
    //
    HWND hwnd = GetSafeHwnd();

    // Create the channel if it has not already been created
    //
    if (ch != NULL) {
          ch = new channelClass((void *)hwnd);
    }

    // Create the mouse if it has not already been created
    // Also, attach the channel to the mouse.
    if (mouse == NULL) {
          mouse = new mouseClass(this);
          mouse->attachChannel(ch);
    }

    CView::OnInitialUpdate();
}
```

The attachment of the channel to the mouse is an important concept in using sensors in Amber. Amber sensors are devices that can apply translation, rotation, and user defined events to either an Amber channel or geometry. For example, you can not only attach a mouse to a channel, but you could also attach a mouse to a geometry so that the mouse sensor could be used to spin or translate it. This will be demonstrated further in this lesson.

In order for us to translate the universe at a reasonable rate, we will set our translation speed to be 10% of the universe size for every frame that is rendered. You can adjust this number to suit the rendering speed of your platform. After the mouse creation code of the **OnInitialUpdate()** function, add the following code:

```
    // Create the mouse if it has not already been created
    // Also, attach the channel to the mouse.
    if (mouse == NULL) {
          mouse = new mouseClass(this);
          mouse->attachChannel(ch);

    }
    // Adjust the mouse sensor's linear scaling
    // to 10% the size of the universe every frame.
    //
    Vres radius;
    universeClass *univ = amber->getCurrentUniverse();
```

```
    radius = univ->getRadius();
    mouse->setLinearScaling(radius * 0.1);
```

Now that we have added the mouse, we must make sure that it is properly deleted upon exit from the application. Amber does not manage sensors. The user must explicitly delete all sensors that they create. In the same "lessovw.cpp" file, add the following lines to the class destructor:

```
Clesson2View::~Clesson2View()
{

    // Delete mouse
    delete mouse;
}
```

We can now compile the application and run it. At this time, the user should read the chapter in the Amber Reference Manual on the **mouseClass** sensor. This chapter will describe the proper use of the sensor. Once you are comfortable with navigating the space, continue with the next part of the lesson.

Part 2 - Changing Universe Attributes

On some occasions you may wish to change the attributes of the universe that is being rendered. As an example, we will be altering the universe background color as well as the way in which objects are rendered by Amber. Once again edit the "lessovw.cpp" file and add the following code lines to the **myInit** function:

```
void myInit()
{
    cubeClass *cube;
    V3 pos;
    universeClass *univ = amber->getCurrentUniverse();
    F4 color = {0.0, 0.5, 0.5, 1.0};

    // Change the universe background color.
    //
    univ->setBackgroundColor(color);

    // Change the way in which polygons are rendered
    // to be unfilled.
    //
    univ->setPolyFill(_NOFILL);

    // Create a unit cube
    //
    cube = new cubeClass(1,1,1);

    // Set the cube's position 10 units
    // away from the viewpoint.
    //
    V3_set(pos, 0.0, 0.0, -10.0);
    cube->setPosition(pos);

    // Rotate the cube so that we can get
    // some perspective without moving
    //
```

```
        cube->rotate(Y, 45*DEG_TO_RAD);
        cube->rotate(X, 20*DEG_TO_RAD);

    }
```

Compile, link and run the program. Notice that the background color of the universe has changed and the cube is being rendered as unfilled polygons. To return the polygon rendering to its previous state, simple comment out the line:

```
        univ->setPolyFill(_NOFILL);
```

Part 3 - Attaching The Mouse To Objects

Instead of attaching a channel to a mouse sensor, we can also attach a geometry. To try this, first comment out the following line :

```
    mouse->attachChannel(ch);
```

After this line, add the following lines:

```
        // Create the mouse if it has not already been created
        // Also, attach the channel to the mouse.
        if (mouse == NULL) {
              mouse = new mouseClass(this);
              //mouse->attachChannel(ch);
              cubeClass *g;
              V3 pos;
              F4 red = {1.0, 0.0, 0.0, 1.0};
              g = new cubeClass(0.25, 2.0, 0.25);
              V3_set(pos, 0.0, 0.0, -10.0);
              g->setPosition(pos);
              g->setAllColor(red);
              mouse->attachGeometry(g);
        }
```

After compiling and linking, you will now notice that a red cubic rod has appeared inside the main cube. Also, when you move the mouse, the rod will move accordingly. You can also constrain the sensor so that certain values will not be used. For our example we will constrain all linear motion of the sensor. To do this, add the following lines to the previous ones:

```
        // Create the mouse if it has not already been created
        // Also, attach the channel to the mouse.
        if (mouse == NULL) {
              mouse = new mouseClass(this);
              //mouse->attachChannel(ch);
              cubeClass *g;
              V3 pos;
              F4 red = {1.0, 0.0, 0.0, 1.0};
              g = new cubeClass(0.25, 2.0, 0.25);
              V3_set(pos, 0.0, 0.0, -10.0);
              g->setPosition(pos);
              g->setAllColor(red);
              mouse->attachGeometry(g);
```

```
        // Constrain all linear motion
        //
        mouse->setPosConstraint(X);
        mouse->setPosConstraint(Y);
        mouse->setPosConstraint(Z);
}
```

When you run the program now you will notice that the rod only rotates. It will not move along any linear axis.

Please proceed with Lesson 3 where you will learn to add behaviors to objects within an Amber application.

Lesson 3 - Adding Behaviors

Lesson 3 expands upon the points learned in Lessons 1 & 2 to create a universe with objects that inherit a simple behavior.

In Amber, behaviors fall into three categories. These categories are defined below:

Actions	-	Behaviors specific to an instance of an object in the universe.
Styles	-	Behaviors specific to a class of objects in the universe.
Laws	-	Behaviors applied to all law based objects in the universe.

Part 1 - Adding Styles

In this lesson, we will demonstrate the use of styles by creating a new class. This class will cause a cube to spin about a specified axis. The class header definition ("spin.hpp") is:

```
#ifndef _SPIN_HPP
#define _SPIN_HPP

#include "style.hpp"
#include "shapes.hpp"

class spinClass : public styleClass,  public cubeClass {

public:

  // The local data for our class
  Vres spinRate;
  int    spinAxis;

  // This function must be here to utilize the styleClass since the
  // styleClass defines it as virtual.
  //
  void style(void);

  spinClass(Vres rate, int axis,
            Vres cubeX, Vres cubeY, Vres cubeZ);
  ~spinClass();

};
#endif
```

The class implementation ("spin.cpp") is:

```
#include "spin.hpp"

// The style function which will run every frame
//
void spinClass::style(void) {
    rotate(spinAxis, spinRate);
}
```

```
// Class constructor
//
spinClass::spinClass(Vres rate, int axis,
                                    Vres cubeX, Vres cubeY, Vres
cubeZ) : styleClass (1,1),

cubeClass(cubeX, cubeY, cubeZ)
{
  spinRate = rate;
  spinAxis = axis;
}

// Class destructor
//
spinClass::~spinClass()
{
}
```

After you add this new class file to the project, be certain that precompiled headers are not used on this file. This can be disabled through the "Settings" option of the "Project" menu.

Amber styles provide a means of defining a behavior that can be inherited by a class of objects. In our example, the behavior is spinning. Here we use a generic cube as the object. This could be replace by the **geometryClass** so that any geometry could be loaded to inherit this behavior. If this behavior is to be inherited with others to form a final derived type, be certain that the object that is acted on is defined as **virtual** in the class definition. For example, slightly altering this class allows it to be inherited with others to create a more complex behavior. To do this, alter the class definition to be:

```
class spinClass : public styleClass,  virtual public geometryClass {
```

The **geometryClass** object is then defined in the final derived type. This allows all styles to utilize the same **geometryClass** object. For more information, consult the C++ help files or a C++ manual.

Once this style has been added and compiled into the project, add the following lines to the top of the "lessovw.cpp" file:

```
// lessovw.cpp : implementation of the Clesson3View class
//

#include "stdafx.h"
#include "lesson3.h"

#include "lessodoc.h"
#include "lessovw.h"

#include "amber.hpp"
#include "spin.hpp"

// Make the cubes global so that they can be deleted
//
spinClass *cubes[5];
```

Now change the **myInit** function to be:

```
void myInit()
```

```
{
    V3 pos;
    int i;

    // Create all spinning cubes with random
    // spin axis.
    //
    for (i=0; i<5; i++) {
          cubes[i] = new spinClass( 0.17,
                                    (rand() % 3),
                                    1,1,1);
          // Set them in random positions
          //
          pos[X] = (Vres)(rand() % 10) - 5.0;
          pos[Y] = (Vres)(rand() % 10) - 5.0;
          pos[Z] = (Vres)(rand() % 10) - 5.0;
          cubes[i]->setPosition(pos);
    }
}
```

This initialization function will create five spinning cubes at random locations. The spin axis of each cube is randomized and their spin rate is set at 0.17 radians per frame. This is a simple style that represents the way in which behaviors can be inherited from a class definition.

Part 2 - Adding Actions

The next thing we will do is add a unique function to one of the cubes. This function will be an "action" behavior for the unique object instance. The function is defined as:

```
void randomColorFunc(void *obj)
{
    geometryClass *g = (geometryClass *)obj;
    F4 col;
    col[0] = (float)(rand() % 10)/10.0;
    col[1] = (float)(rand() % 10)/10.0;
    col[2] = (float)(rand() % 10)/10.0;
    col[3] = 1.0;

    g->setAllColor(col);
}
```

This function will randomize an object's color every time it is called. To apply this "action" behavior to one of the objects, add the following line to the **myInit** function:

```
void myInit()
{
    V3 pos;
    int i;

    // Create all spinning cubes with random
    // spin axis.
    //
    for (i=0; i<5; i++) {
          cubes[i] = new spinClass( 0.17,
```

```
                                    (rand() % 3),
                                    1,1,1);
            // Set them in random positions
            //
            pos[X] = (Vres)(rand() % 10) - 5.0;
            pos[Y] = (Vres)(rand() % 10) - 5.0;
            pos[Z] = (Vres)(rand() % 10) - 5.0;
            cubes[i]->setPosition(pos);
    }

    // Add an "action" behavior to the
    // first cube.
    //
    cubes[0]->setAction(randomColorFunc, 10);
}
```

Now when the application is compiled, linked, and run, we see that one of the cubes performs its spinning "style" behavior but also changes its color randomly.

Part 3 - Adding Laws

For this demonstration, we will add a new class that is based on the **lawClass**. All objects derived from the **lawClass** may have universe law's applied to them. Once you define an object as being based on the lawClass, any law that is added to the universe using the **universeClass::addLaw()** function will be applied to this object. If the object has a visual component (geometry), the visual component will position itself according to the updated kinematic information in its base class.

First we must define a new class of object that has the **lawClass** at its base. The header definition is "spingrav.hpp":

```
#ifndef _SPINGRAV_HPP
#define _SPINGRAV_HPP

#include "law.hpp"
#include "spin.hpp"
#include "kine.hpp"

class spinGravClass : public spinClass,
                      public lawClass,
                      public virtual kinematicClass {

public:

    spinGravClass();
    ~spinGravClass();

};

#endif
```

Notice that this class inherits the properties of the **spinClass**. You must also declare a **kinematicClass** inheritance at the most derived class. This is because the most derived class actually must reference the constructor of any **virtual** base class.

The class implementation ("spingrav.cpp") is:

```
#include "spingrav.hpp"


spinGravClass::spinGravClass() : spinClass(0.02, 0.5, 0.5, 0.5, 0.5),
                                 lawClass(),

kinematicClass()
{
    setGeo(this);
}

spinGravClass::~spinGravClass()
{
}
```

After you add this new class file to the project, be certain that precompiled headers are not used on this file. This can be disabled through the "Settings" option of the "Project" menu.

To use this new class, edit the "lessovw.cpp" file and add the following lines:

```
#include "amber.hpp"
#include "spin.hpp"
#include "spingrav.hpp"
#include <math.h>

// Make the cubes global so that they can be deleted
//
spinClass *cubes[5];
spinGravClass *cubesG[5];
```

You must also add a new initialization function **myInit2**:

```
void myInit2()
{
    V3 pos;
    int i;

    // Create all spinning gravity cubes.
    //
    for (i=0; i<5; i++) {
          cubesG[i] = new spinGravClass();
          pos[X] = (Vres)(rand() % 10) - 5.0;
          pos[Y] = (Vres)(rand() % 10) - 5.0;
          pos[Z] = (Vres)(rand() % 10) - 5.0;
          cubesG[i]->setPosition(pos);

          // Set kinematic pos.
          cubesG[i]->setPos(pos);
    }
}
```

Now define a law that will be applied to all **lawClass** based objects. The law for this demonstration is a point gravity source:

```
void pointGravity(lawClass *obj) {

    V3 accel;

    // Get the position of the object
    //
    V3_copy(accel, obj->kinematicClass::position);

    // Create acceleration vector
    V3_multk(accel, -0.25);

    obj->setAccel(accel);
}
```

The final step is to make a call to the **myInit2** function and add the newly defined law to the universe. To do this, add the following lines to the **OnInitialUpdate** function:

```
void Clesson3View::OnInitialUpdate()
{
    // TODO: Add your specialized code here and/or call the base
class

    // Retrieve the HWND pointer to this window
    //
    HWND hwnd = GetSafeHwnd();

    // Create the channel if it has not already been created
    //
    if (ch == NULL) {
          ch = new channelClass((void *)hwnd);
          myInit();
          myInit2();

          universeClass *univ = amber->getCurrentUniverse();
          univ->addLaw(pointGravity);
    }
```

After compiling, linking, and running, you will notice that there are five more smaller cubes that are oscillating around a central gravity point. To make the display a bit more interesting, add an initial random velocity to each **spinGravClass** object. Add the following lines to the **myInit2** function:

```
void myInit2()
{
    V3 pos;
    int i;
    V3 vel;

    // Create all spinning gravity cubes.
    //
    for (i=0; i<5; i++) {
          cubesG[i] = new spinGravClass();
          pos[X] = (Vres)(rand() % 10) - 5.0;
          pos[Y] = (Vres)(rand() % 10) - 5.0;
          pos[Z] = (Vres)(rand() % 10) - 5.0;
          cubesG[i]->setPosition(pos);
```

```
            // Set kinematic pos.
            cubesG[i]->setPos(pos);

            // Add initial random velocity
            //
            vel[X] = ((Vres)(rand() % 10) - 5.0)/5.0;
            vel[Y] = ((Vres)(rand() % 10) - 5.0)/5.0;
            vel[Z] = ((Vres)(rand() % 10) - 5.0)/5.0;
            cubesG[i]->setVel(vel);
        }
    }
```

This concludes the demonstration of actions, styles, and laws. Feel free to alter the source code and experiment with changing values of each behavior type.