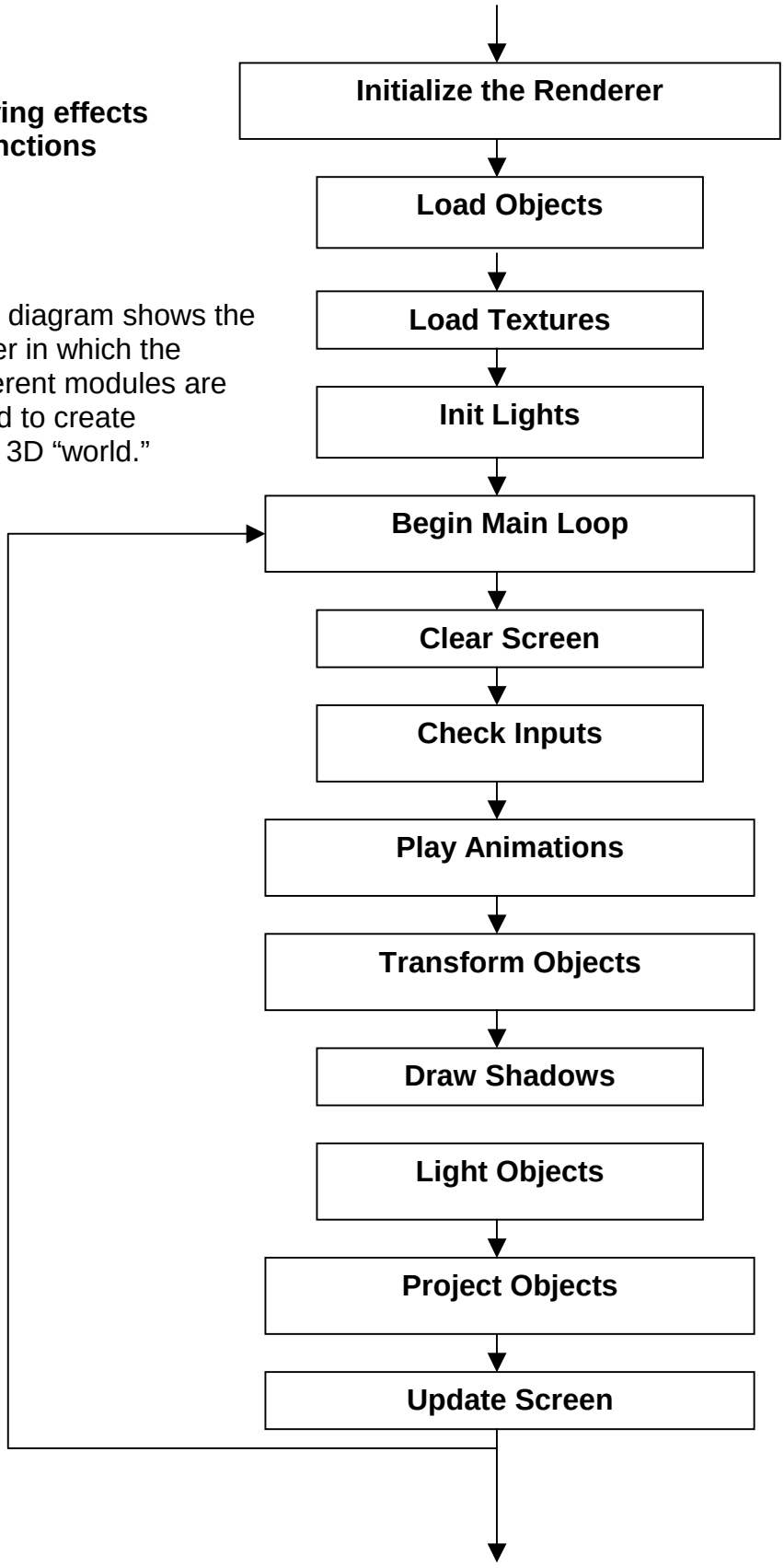
User's Guide

Implementing and displaying effects using the 3D Engine's Functions

The diagram shows the order in which the different modules are used to create the 3D "world."



The following sections explain

How to use…..

The Renderer

Handle User Input

Use Sound Effects

BillBoard

The Camera

Lights

Particle Effects

Objects

Shadows

Textures

How to Use ....*The Renderer*

Creating

The PW3D class owns all of the cool 3d effects functions. An instance of the PW3D class is already created upon entering your program. The variable name is “pw3d” and is available to any file that #include’s “pw3d.h”.

Displaying

In order for the renderer to function properly, the following MUST occur:

1. Call **pw3d.GetTextures()** for all textures that you wish to use.
2. Add lights to the scene using **pw3d.Add<typeoflight>()**
3. Call **pw3d.Cls()** to clear the screen BEFORE rendering.
4. Call **pw3d.CheckEvents()** at some point in your main loop in order to capture keyboard/mouse input.
5. After calling **pw3d.Render()** and **pw3d.Blit()**, call **pw3d.ScreenSwap()** to swap the buffers and to make graphical elements actually display.
6. Finally, call **pw3d.Render()** only AFTER all graphical constructs have been added, modified, or updated in order to display them.

NOTE:
pw3d.PlaySound() can be called anytime. It is independent of **pw3d.Cls()** and **pw3d.ScreenSwap()**. Also, all **pw3d.GetXInput()** functions can be called at any time, but they will only be correct up to the last time that pw3d.CheckEvents() was last called.

Functions

void ScreenSwap() – Swap the back buffer to the screen. This enables all of the rendered pictures to be displayed on the monitor.

void SetDrawPoints(bool dpoints) – Tell the renderer to draw the vertices of the objects.

void SetDrawTexture(bool dtext) – Set the renderer to draw objects with textures.

void SetDrawLines(bool dlines) – Set the renderer to draw the lines connecting the vertices of each triangle.

void AddObjectToPipeline(Object * obj) – Add the object obj to the rendering pipeline. All objects in the pipeline are drawn to the backbuffer with a call to Render(), and are then removed from the pipeline.

void AddPEffectToPipeLine(PEffect * p) – Add a particle effect to the pipeline. All particle effects in the pipeline are drawn to the backbuffer with a call to **Render()**, and are then removed from the pipeline.

void AddBillBoardToPipeline(BillBoard * b) – Add a Billboard to the pipeline. All Billboards in the pipeline are drawn to the backbuffer with a call to **Render()**, and are then removed from the pipeline.

void Cls() – Clear the backbuffer for clean rendering. This should be done after each call to **ScreenSwap()**.

void GetTexture(char * bmpname, int width, int height, int texturenum) – Loads a texture into memory of size *width x height*. When this function is called, it is given the number texturenum. This number is later used when assigning textures to objects.

void LightObjects() – Light up all objects in the pipeline.

void RenderAllGL() – This function is a test function that is not meant to be used in the actual product. It renders each object using OpenGL transformation routines.

void Render() – Render is the main call of the renderer. It tells the renderer to draw all objects, particle effects, and billboards to the backbuffer.

void RenderAsShadows(float red, float green, float blue, float transparency) – Tells the renderer to draw each object in the pipeline to the screen in a constant color with a set transparency. This is used if each object in the pipeline has been projected as a shadow, and then must be drawn as a shadow.

void RenderObjectAt(Object * obj, int x, int y) – The intended purpose of this function is to render an object at a screen coordinate instead of in World-Space. It is currently broken and doesn't function correctly.

void Blit(int texturenum, int x, int y) – Blit a 2D bitmap to a screen coordinate. The bitmap must be loaded as a texture before it can be blitted to the screen. Currently, any black color in the bitmap (0,0,0) will be drawn as transparent. There is a known bug on 16 bit video cards however, the blitted images turn out white. This is most likely due to the way the bitmap is loaded.

int CheckEvents(void) – Update all of the input variables to key presses, mouse input, etc.

unsigned int Timer(void) – Return the current number of seconds since windows was started. To use it, do something like so:

```
unsigned int nexttime = pw3d.Timer() + 1000;
while(nexttime > pw3d.Timer()) {}  //wait for 1 second
```

void Text(char *mesg, int xp, int yp) – This function should only be used for debugging purposes as it flickers when you output text. The way it works is you specify a string mesg, and a screen location xp,yp and it will output the text to the screen.

int ScreenWidth() – Return the width, in pixels, of the screen.

int ScreenHeight() – Return the height, in pixels, of the screen.

How to … Handle User Input

User input is handled through Microsoft DirectInput. The DirectInput system is initialized by default at the time the window is created for the rendering to be done. It allows the programmer to handle keyboard and mouse input quickly to meet the requirements of today's entertainment software, known in some elite circles as games.

Capturing Mouse and Keyboard Input

The **CheckEvents()** function MUST be called somewhere within the main loop of your program in order for you to capture any mouse and/or keyboard input. It is called like so:

```
pw3d.CheckEvents();
```

Additional Mouse Functions

Additional functions within pw3d provide the ability to SetMouseSpeed(), find out if the right or left mouse button was clicked or is down, get the X or Y mouse position, and determine the mouse acceleration along the X or Y axis.

int MouseX(void) – Return the x-coordinate of the mouse position.

int MouseY(void) – Return the y-coordinate of the mouse position.

int MouseAccelX(void) – Return the amount the mouse x position has changed in the x direction. This value is positive if the mouse moved right, negative if the mouse moved left.

int MouseAccelY(void) – Return the amount the mouse y position has changed in the y direction. This value is positive if the mouse moved down, negative if the mouse moved up.

int MouseLMB(void) – Return's 1 if the left mouse button is down.

int MouseRMB(void) – Returns 1 if the right mouse button is down.

int MouseClickL(void) – Returns 1 if the left mouse button was clicked.

int MouseClickR(void) – Returns 1 if the right mouse button was clicked.

void SetMouseSpeed(float speed) – Sets the mouse speed to a multiple of float. The value of 1 is default, to make the mouse move slower set the speed to < 1.0f. To make the mouse move faster set the speed to > 1.0f.

Additional Keyboard Functions

Additional functions within pw3d provide the ability to get the last key pressed, see if the key flag is down, find out if a particular key is down, or look to see if specifically the ESC key is down.

int Inkey(void) – Return the last key pressed in the form of a keyboard scan code. For a list of all scan codes, check the **pw3d.h** file.

int KeyFlags(void) – Return what function keys are being pressed, such as shift keys or control keys. This does not need to be explicitly called by the user.

int KeyFlags(int flag) – Returns 1 if keyflag **flag** is down.

int KeyDown(int sCode) – Returns 1 if the key with scan code sCode is currently down (or rather was when the last CheckEvents was called).

int KeyDown_ASCII(int ch) – Returns 1 if the key corresponding to the ASCII character **ch** is down.

int KeyDownESC(void) – Returns 1 if the escape key is down.

How to use … Sound Effects

Sound Effects in PW3D are done through the use of Microsoft's DirectSound API. GameKit users will notice a striking similarity between PW3D sound implementation and GameKit sound implementation. The DirectSound system is initialized automatically by the initial setup process. You do not need to do anything explicit to begin working with sound.

Creating

1. *Loading .WAV files.*

   .WAV files must be loaded before they can be played. To do this, call

   pw3d.LoadWAV(integernameofsoundbuffer, WAVfilename)

   on each .WAV file before trying to play it using the pw3d.PlaySound function.

2. *Playing Sound Files*
   To play a sound file, call either the PlaySound or PlaySoundLooping function.
   a. **pw3d.PlaySound(integernameofsoundbuffer, integer-left-percent, integer-right-percent)**
   b. **pw3d.PlaySoundLooping(integernameofsoundbuffer, integer-left-percent,integer-right-percent)**

3. *Stopping the sound*.
   To stop a sound from playing that has been started using the PlaySoundLooping function, call the StopSound function using

   pw3d.StopSound(integernameofsoundbuffer)

4. *Changing the Sound Volume*
   Call the SetSoundVolume function.

   pw3d. SetSoundVolume(integernameofsoundbuffer)

5. *Verifying that you have loaded a sound.*
   Calling the CheckValidSoundRange function will allow you to verify that your sound has loaded correctly.
   a. **pw3d.CheckValidSoundRanged(integernameofsoundbuffer, integerSuppressWaring)**

How to Use ….*BillBoard*

Creating

1. Declare a Billboard object using the following declaration:
    a. **BillBoard a_billboard(leftcoordinate,rightcoordinate,topcoordinate, bottomcoordinate);**

    The constructor takes the coordinates of the BillBoard in local space as parameters.

Moving

2. Next, move the BillBoard to its desired location using either the MoveTo() or MoveBy() function.
    a. **a_billboard.MoveTo(float_absoluteXcoordinate, float_absoluteYcoordinate);**
    b. **a_billboard.MoveBy(float_offsettoXlocation, float_offsettoYlocation);**

Displaying

3. Add a texture to the BillBoard using the SetTextureNum function

    a_billboard.SetTextureNum(numberofloadedtexture);

4. Orient the BillBoard with respect to the camera by calling the **Orient()** function.

    a_billboard.Orient(float_pitch, float_yaw);

5. Enable the BillBoard to be displayed by adding it to the rendering pipeline using the following:

    pw3d.AddBillBoardToPipeLine(&a_billboard);

6. The next call to **pw3d.Render()** will render the BillBoard on the screen.

Miscellaneous

1. Find out the texture number that a_billboard has using the **GetTextureNum()** function.

    an_integer = a_billboard.GetTuxtureNum();

2. Other commands will retrieve select X and Y corner coordinates of the BillBoard. Please see the billboard.h file for more information.

How to Use ….*The Camera*

Creating

The Camera does not need to be created and initialized by the User. The Camera is initially positioned at (0, 0, 0) and is pointing down the positive Z axis.

Displaying the World

The 3d world is viewable only through the eye of the Camera. Move the camera and display the world using the Camera Movement functions found in the PW3D object:

Functions

pw3d.PointCameraAt(float Xcoord, float Ycoord, float Zcoord) – Orients the camera so that it points at a certain point in World-Space.

pw3d.MoveCameraBy(float Xcoord, float Ycoord, float Zcoord) – Move the camera around in World Space by Offsets.

pw3d.MoveCameraTo(float Xcoord, float Ycoord, float Zcoord) – Move the camera around in World Space to a certain x,y,z coordinate.

pw3d.MoveCameraForward(float dist) – Move the camera forward in World Space by the distance passed in.

pw3d.MoveCameraBackward (float dist) – Move the camera backward in World Space by the distance passed in.

pw3d.MoveCameraLeft(float dist) – Move the camera left in World Space by the distance passed in. This is the same as Side-Stepping in a First Person Shooter.

pw3d.MoveCameraRight(float dist) – Move the camera right in World Space by the distance passed in. This is the same as Side-Stepping in a First Person Shooter.

pw3d.MoveCameraUp(float dist) – Move the camera up in World Space by the distance passed in.

pw3d.MoveCameraDown(float dist) – Move the camera down in World Space by the distance passed in.

pw3d.TurnCameraRight(float radians) – Turn the Camera right by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

pw3d.TurnCameraLeft(float radians) – Turn the Camera left by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

pw3d.TurnCameraUp(float radians) – Turn the Camera up by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

pw3d.TurnCameraDown(float radians) – Turn the Camera down by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

pw3d.RollCameraRight(float radians) – Roll the Camera right by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

pw3d.RollCameraLeft(float radians) – Roll the Camera left by the radians passed in. **Note**: 360 degrees is equivalent to 2*PI radians.

How to Use ....*Lights*

Creating

1. All light manipulation functions are found in the pw3d class. *Create* a bulb light, spot light, ambient light, or directional light object using one of the following function calls giving it an ID number, an initial x-y-z location, and initial direction, r-g-b color values, range, and/or intensity depending on the type of light created. Intensity values range from 0.0 to 1.0 . Color values range from 0.0 to 1.0 .

    a. **pw3d.AddBulbLight(idNum, initialXpos, initialYpos, initialZpos, lightRange, red, green, blue, intensity);**

    b. **pw3d.AddAmbientLight(idNum, red, green, blue, intensity);**

    c. **pw3d.AddSpotLight(idNum, initialXpos, initialYpos, initialZpos, Xdir, Ydir, Zdir, red, green, blue, intensity);**

    d. **pw3d.AddDirectionalLight(idNum, Xdir, Ydir, Zdir, red, green, blue, intensity);**

Manipulating

1. To *move lights* around, use the MoveLightBy and MoveLightTo functions. MoveLightTo moves the light to the specified X,Y,Z coordinates. MoveLightBy moves the light by the X,Y,Z offset specified relative to it's current location. To use these functions, pass in the idNum of the light you wish to move and specify the position (if MoveLightTo) or the offset (if MoveLightBy) .
    e. **MoveLightTo(idNum, newXpos, newYpos, newZpos);**
    f. **MoveLightBy(idNum, movebythismuchinXdir, movebythismuchinYdir, movebythismuchinYdir);**

    Ex. pw3d.AddBulbLight(1, 20, 300, 100, 1.0, 1.0, 1.0, .75);
    pw3d.MoveLightTo(1, 100, 100, 100);
    pw3d.MoveLightBy(1, 10, 14, 25);

    This sequence of functions adds a bulb light to the seen, moves it to a location, and then moves it by a certain offset.

2. The **SetLightColor()** function allows you to *change the light's color* on the fly after the light has been created.

    pw3d.SetLightColor(idNum, newRed, newGreen, newBlue);

    Ex. pw3d.SetLightColor(1, .43, .75, .65);

3. The SetLightIntensity function allows you to change the light's intensity at-will after the light has been created.

pw3d.SetLightIntensity(idNum, newIntensity);

Ex. pw3d.SetLightIntensity(1, .75);

4. If you want to point a light at a specific location no matter where the light is placed in the scene, use the PointLightAt function.

   pw3d.PointLightAt(idNum, pointatX, pointatY, pointatZ);

   Ex. pw3d. PointLightAt (1, 50, 25, 10);

5. If you want to point a light in a specific direction no matter where the light is placed in the scene, use the function.

   pw3d.SetLightDirection (idNum, pointindirX, pointindirY, pointindirZ);

   Ex. pw3d.SetLightDirection (1, 50, 25, 10);

6. If you want to turn a light ON or OFF, call the TurnOnLight or TurnOffLight function.

   pw3d.TurnOnLight (idNum); // sets intensity to 1.0 – full on
   pw3d.TurnOffLight(idNum); // sets intensity to 0.0 – full off

   Ex. pw3d.TurnOnLight (1);
   pw3d.TurnOffLight (1);

Displaying

1. Once the lights have been created, they are automatically added to the rendering pipeline. All you have to do is call the LightObjects function to enable the Renderer to display them on the screen.

   pw3d.LightObjects();

   This displays all the lights which have been created up to this point. However, all light manipulations must have been done prior to calling this function or the adjusted light will not be displayed.

How to Use ….*Objects*

Creating

While Objects can be created with your own custom-built function calls, this is not recommended as it is rather complicated. Using the following steps, Objects can be easily created and displayed.

1. Create the Object. (See the file object.h for details.)

   The best way to create an object is to load one from a .pob file like so

   Object torch("torch.pob");

   You can also create custom objects by passing in an array of vertices, triangles, texture coords, and animations like so:

   ```
   Object an_object(  numberof vertices,
                      numberoftriangles,
                      arrayofvertices,
                      arrayoftriangles,
                      arrayoftexturecoordinates,
                      numberofanimations,
                      arrayofanimations );
   ```

2. Move, Rotate, Transform, or otherwise manipulate the Object using functions found in the object.h file.

   For example, you can move an object to a position in World Space by calling:

   an_object.MoveTo(0, 100, 0);

   Then you can rotate it to turn by 90 degrees to the right:

   an_object.RotateBy(0, PI/4, 0);

   Then you can have it play it's animation:

   an_object.PlayAnimation(1); //play animation 1

3. Call the Object's **Transform()** function to cause these manipulations to take effect.

4. Call **pw3d.LightObjects()** to light up all the Objects in the scene.

Displaying

5. Call **pw3d.Render**() to render all Objects in the pipeline.

*How to Use ….**Shadows***

Simple Shadows

Creating

To create a simple shadow you must first load in a texture to be used as that objects shadow map. Chose the shadow map you use well. For instance, if you are projecting the shadow of a table, use a square shaped map. If you are trying to project the shadow of a person, use a round map. Shadow maps should be brightest where you wish them to be darkest in the project. An example of a square shadow map is as follows:

To initialize a shadow map you must first load the texture.

pw3d.GetTexture("BitchinCamaro.bmp", 64, 64, SHADOW_MAP);

Second, you must enable simple shadows on the object you wish to project the shadow to.

object.DrawSimpleShadows(TRUE);

Third, you must tell the object which shadow texture to use.

object.SetShadowTexture(SHADOW_MAP);

Finally, you must tell the height on the y-axis with witch the shadow is drawn.

object.SetSimpleShadowHeight(1);

The renderer will then automatically draw the simple shadow after the object is rendered from the pipeline. The shadow will mold itself to match the first directional light found in the lighting system.
Creating a cool effect like a bulb light moving around the scene and causing shadows to respond to it is fairly simple. Add a directional light to be light 0 pointing in an arbitrary direction with 0 intensity.

pw3d.AddDirectionalLight(0, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 1.0f, 0.0f);

Next add a bulb light at some arbitrary position with the color you wish.

pw3d.AddBulbLight(1, 0,0,0, 180.f, 1.0f, 1.0f, 1.0f, 1.0f);

Now, move the bulb light around, while changing the original directional light's direction. It will seem to the viewer that the bulb light is interacting with the shadows!

```
pw3d.MoveLightTo(2, dist*cos(radians), 100, dist*sin(radians));
pw3d.SetLightDirection(0, -5*cos(radians), -1, -5*sin(radians));
radians += .01;
```

Note: The above code causes the bulb light to move in a circle on the x/z plane at a height of 100, while changing the direction of the directional light to accommodate the position of the bulb with respect to the objects being shadowed.

Complex Shadows

Creating

Ironically, complex shadows are actually simpler to implement than simple shadows. They do take more processing however and are a little less intuitive.

To create a complex shadow, you must first draw your object in the standard manner.

After the first call to pw3d.Render(), you must project the shadow to a plane. This is done by called the objects ProjectToPlane() function.

object.ProjectToPlane(float a, float b, float c, float d, float dx, float dy, float dz);

The ProjectToPlane function squishes each of the objects vertices to the plane defined by **ax + by + cz = z**. It projects the plane with respect to light direction dx,dy,dz. (note, the x/z plane is 0,1,0,0)

In order for these projections to take effect, you must call the object's **Transform()** function again.

object.Transform();

After all of your objects are projected to the plane you wish them to be drawn on, you must call the **RenderAsShadows()** function in the renderer.

pw3d.RenderAsShadows(float red, float green, float blue,

float transparency);

This function causes everything you just added to the rendering pipeline to be drawn in the color you passed in, with the transparency you passed in.

How to Use ….*Particle Effects*

Creating

1. Declare an object of type PEffect with a starting location and a texture number. The "standard" particle effect texture is a small lens flare, texture number 4. The function call has the form:

   PEffect a_particle(intitialXposition, initialYposition, initialZposition)

   Ex. PEffect a_particle(100,150, -200, 4);

   You may either give the particle an initial location or create it at 0,0,0 and move it to where you want it generated before calling the InitEffect function.

   Ex. PEffect a_particle(0, 0, 0, 4);
   A_particle.MoveTo(100, 150, -200);

2. Call the InitEffect function with the particle dimensions, r-g-b colors for the particle, and the type of effect that you want (fire, smoke, particle, fairydust, fog). Ideally, the r-g-b values should be between 0.0 and 1.0 . If they are larger than 1.0, however, the program will take care of it. The function call has the form:

   a_particle(width, depth, height, red, green, blue, effect_type)

   Ex. a_particle.InitEffect(5.0, 5.0, 5.0, 0.2, 0.8, 0.2, fire)

Displaying

1. **Call the PEffect Update function**
   Ex. a_particle.Update();

2. **Add the PEffect object to the rendering pipeline**
   Ex. Pw3d.AddPEffectToPipeline(&a_particle);

Custom Effects

Follow the previous steps, then call the other initialize routines to select the particles direction and dispersion rate.

How to Use ....*Textures*

Creating

1. All textures which are to be used must be loaded into video memory before they can be used within the program. The GetTexture function must be called for each and every texture which you wish to use. Pass the name of the texture's bitmap, it's width, height, and a desired textureIDNumber.

    pw3d.GetTexture(texturebitmapname, integerwidthinpixels, integerheightinpixels, user-definedintegertextureIDNumber);

    Ex. pw3d.GetTexture(AStarTextureBitmap, 14, 14, 4);

Displaying

1. To display a texture, simply place the texture IDNum that you assigned it into the idNum parameter of any function call requiring a texture ID number in its parameter list. The PEffect constructor is one such function call which requires a texture number. If I wanted to use the above texture in a PEffect, I would pass it the textureIDNumber of for which I assigned to AStarTexture by calling the GetTexture function.