This application demonstrates how to install a filter in an OpenGL pipeline using Objective-C MVC design pattern. Here our model in the MVC design pattern is the filter object encapsulating the OpenGL algorithmic workflows of loading, compiling, linking and binding of a fragment shader to a program object; data stream generator; texture initialization; framebuffer object instantiation.

We generate new data for our array (texture), and therefore our computation, by rotating 3D models. In this application, and to demonstrate the separation between data generation on the CPU side, and processing of data on the GPU side, we shall use the immediate mode functionality of OpenGL to generate data. Here, this has been encapsulated in an object that maintains a display list for three tori and the *Stanford Bunny*, and the current angle of rotation. To be certain this data can be generated on the GPU side using VBOs and PBOs.

Furthermore, in this application and with the MVC design pattern, the controller object acts as a request broker between the filter and the view, thus feeding the data into the filter and visualizing the results of the computation by updating the OpenGL view.

Additionally, this application will demonstrate how to use dictionaries to encapsulate the relevant data for initialization of OpenGL view's pixel format attributes, and uniforms associated with a GLSL fragment program or a computational kernel.

To initialize a filter we start by setting a texture size,

```
textureSize = NSMakeSize( 1024, 1024 );
```

where `textureSize` is of type `NSSize`. Here our texture size, or an array, will be of size `1024 x 1024`. Next we shall initialize a data bounds of type `NSRect` to be

```
dataBounds = NSMakeRect( -1, -1, 1, 1 );
```

that is, values for our orthographic projection 2D range.

Now we shall initialize a dictionary for the uniform associated with our fragment shader

```
NSArray      *theUniformKeys = [NSArray arrayWithObjects:@"textureUnit", nil];

NSArray      *theUniformValues = [NSArray arrayWithObjects:
                                     [NSNumber numberWithInt:0], nil];

NSDictionary *theUniformDictionary = [NSDictionary
                                     dictionaryWithObjects:theUniformValues
                                     forKeys:theUniformKeys];
```

and thus initializing the uniform `"textureUnit"` with the value `0`.

To instantiate a filter object with the `"LaplacianEdgeDetectionFilter"` computational kernel (fragment shader), uniforms, data domain (orthographic projection 2D range), and data capacity (texture size), we shall invoke the designated initializer,

```
filter = [[OpenGLFilterUtilityToolKit alloc]
                    initWithFilterInAppBundle:@"LaplacianEdgeDetectionFilter"
                    uniformDictionary:theUniformDictionary
                    bounds:dataBounds
                    size:textureSize];
```

where `filter` is of type `OpenGLFilterUtilityToolKit`. To run the filter, we shall call the public instance method `update`:

```
- (void) update
{
      [filter prepare];
      [models rotate];
      [filter run:dataBounds];
}
```

This method updates the texture by rendering the models (the rotating *Stanford Bunny* and 3 tori) and copying the image to a texture. Furthermore in a second pass we render again by using the texture as input to the filter. The results produced by the filter are then copied to the allocated texture. The resulting texture is then used for visualizing the results of our computation.

A filter in general encapsulates the following 5 concepts:

1. Textures are equivalent to arrays that one is normally accustomed to in the standard computational models. Here, one allocates a texture large enough to fit the data:

```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA8,width,height,0,GL_RGB,GL_FLOAT,0);
```

Next we shall be binding our texture to a frame buffer object, through first creating,

```
glGenFramebuffersEXT(1, &framebuffer);
```

and then,

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, framebuffer);

glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
                           GL_COLOR_ATTACHMENT0_EXT,
                           GL_TEXTURE_2D,
                           texture,
                           0 );
```

Specifying `GL_COLOR_ATTACHMENTm_EXT` enables drawing only to the image attached to the frame buffer at `GL_COLOR_ATTACHMENTm_EXT`. Each `GL_COLOR_ATTACHMENTm_EXT` adheres to

```
GL_COLOR_ATTACHMENTm_EXT = GL_COLOR_ATTACHMENT0_EXT + m.
```

The initial value of `GL_DRAW_BUFFER` for application-created frame buffer objects is `GL_COLOR_ATTACHMENT0_EXT`.

2. A fragment program represents a computational kernel applied in parallel to multiple fragments simultaneously. In this application we shall load a computational kernel or a fragment program that performs Laplacian edge detection in real-time.

```
GLhandleARB programObject = glCreateProgramObjectARB();

GLhandleARB fragmentShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

glShaderSourceARB(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShaderARB(fragmentShader);

glAttachObjectARB(programObject, fragmentShader);

glLinkProgramARB(programObject);

GLint progamLinked;

glGetObjectParameterivARB(programObject, GL_OBJECT_LINK_STATUS_ARB, &progamLinked);
```

3. One-to-one pixel-to-texel mapping:

- We need a one-to-one mapping of pixels to texels in order to ensure every element of our texture is processed. By setting our viewport to the dimensions of our destination texture and drawing a screen-sized quad, that is to say,

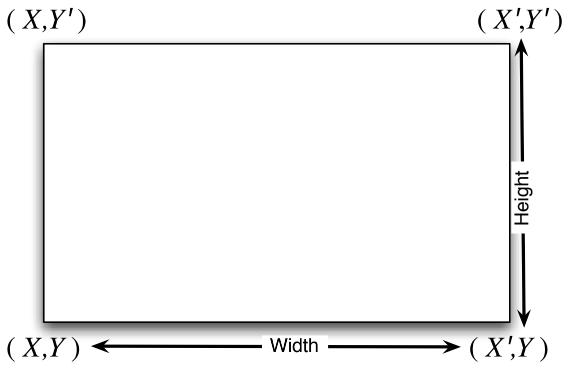
```
glViewport(0, 0, width, height);
```

  one ensures that every pixel of our texel is generated and processed in the fragment program.

- One-to-one pixel to texel mapping is achieved through the use of an orthographic projection matrix. That is to say, here one sets the projection matrix to orthographic in a closed domain defined by

$$\mathbb{D} = \{[X, Y] \times [X', Y']\} = \{[X, Y] \times [X + W, Y + H]\}$$

  where W is the width and H is the height, so that,



```
glMatrixMode( GL_PROJECTION );
glLoadIdentity( );

gluOrtho2D( x, width, y, height );

glMatrixMode( GL_MODELVIEW );
glLoadIdentity( );
```

4. In order to utilize a fragment program in an OpenGL pipeline, it is essential that one should generate pixels. In this application we shall render our models so that the fragment shader (or the computational kernel) processes each data element stored in the target texture. Since we will be using a known set of projection and viewport settings, and by defining the four corner vertices of the quad, we will be transforming each data element into a unique fragment by filling a quad that covers the entire viewport. With the use of normalized texture coordinates `GL_TEXTURE_2D`, this functionality is encapsulated by the following OpenGL calls:

```
glBegin( GL_QUADS );

    glTexCoord2f( 0.0f, 0.0f );
    glVertex3f( x, y, -0.5f );

    glTexCoord2f( width, 0.0f );
    glVertex3f( width, y, -0.5f );

    glTexCoord2f( width, height );
    glVertex3f( width,  height, -0.5f );

    glTexCoord2f( 0.0f, height );
    glVertex3f( x, height, -0.5f );

glEnd();
```

Hence, by assigning texture coordinates as vertex attributes to the four vertices, the four vertices will be mapped into screen space by the vertex stage of the pipeline. The rasterizer will then perform bilinear interpolation for every pixel, interpolating the position of the pixel and the vertex attributes from the vertex data, and so generating a fragment for every pixel covered by the quad. These interpolated values are in turn passed to the fragment shader. Thus, and in terms of a computational process, by rendering a textured quad, the computational kernel (fragment program) is executed for all data elements of an array (or a texture).

5. Once the fragment shader (computational kernel) is applied to a viewport-sized quad, the results of the computation will reside in the frame buffer. That is to say,

    I. Bind to FBO
        A. Enable the FBO
        B. Draw into `GL_COLOR_ATTACHMENT0_EXT`
        C. Set the viewport to the dimensions of our texture
    II. Draw some geometry or model
    III. Copy the results to the FBO bound texture
        A. Draw into the FBO
            1. Draw into `GL_COLOR_ATTACHMENT1_EXT`
            2. Bind to the first texture
        B. Execute the computational kernel
            1. Enable the computational kernel
            2. Generate data in the form of a quad the size of our viewport
            3. Disable the computational kernel
    IV. Display the results
        A. Disable the FBO
        B. Draw into the default draw buffer
        C. Bind to the second texture
        D. Render a full-screen quad textured with the results of our computation

    The texture data can now be used as input for display or additional computations.