How to use `PixelBufferObject`

The startup code includes a very useful class called `PixelBufferObject`, which simplifies communication between C++, CUDA, and OpenGL.

How to load an image into a `PixelBufferObject`:

```
PixelBufferObject * pbo = PixelBufferObject::loadImage("image.png");
```

How to create an empty `PixelBufferObject`:

```
PixelBufferObject * pbo = PixelBufferObject(width, height);
```

How to access a `PixelBufferObject` array from C++:

```
{
   PixelBufferObject::MemoryMap map = pbo->mapHostMemory();
   ... map.getPointer() ...
}
```

When you call `map.getPointer()` you will get a `uchar4` pointer into host memory. You can treat this as a "real pointer" even thought it is really a memory map into GPU memory. Note that it is *very important* to wrap this call in curly braces `{}`. The curly braces introduce a new scope, which will automatically handle starting the memory map (`MemoryMap` constructor) and closing the memory map (`MemoryMap` destructor). I'm not sure what will happen if you have two memory maps open at the same time, but it might not be pretty. So use the curly braces!

How to access a `PixelBufferObject` array from CUDA:

```
{
   PixelBufferObject::MemoryMap map = pbo->mapDeviceMemory();
   some_kernel<<<...>>>(..., map.getPointer(), ...);
}
```

When you call `map.getPointer()` you will get a `uchar4` pointer into device memory. All the same memory map scoping rules apply as with C++ access.

How to display a `PixelBufferObject` using OpenGL:

```
pbo->bindTexture();
glBegin(GL_QUADS);
glTexCoord2f(0.0f,0.0f); glVertex2i(0, 0);
glTexCoord2f(1.0f,0.0f); glVertex2i(w, 0);
glTexCoord2f(1.0f,1.0f); glVertex2i(w, h);
glTexCoord2f(0.0f,1.0f); glVertex2i(0, h);
glEnd();
```