Code Appendix – Reference

Contains the majority of the main classes, besides boilerplate.

Main.cpp

```
#pragma comment (lib, "opengl32.lib")
#pragma comment (lib, "glut32.lib")

#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdlib.h>
#include "GLee/GLee.h"
#include <GL/glut.h>

#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <direct.h>
#include <string.h>

#include "Scene.h"
#include "AllScenes.h"

#include "GameScene.h"

#include "timer.h"
#include "FPScounter.h"

#include <fstream>

int winWidth;
int winHeight;

bool fogSelect,lightSelect,freeCamera,paused;
Scene *currentScene;

double startTime;
double elapsedTime;

bool hasOpenedFile;

Timer timer;
FILE * outputFile;

FPSCounter fpsCounter;

void setupScenes()
{
	currentScene = new SceneTest4();//new GameScene(timer.GetTime());
}
void changeScene(int SceneNum)				//this is the backend for swapping scenes
{
	switch(SceneNum)
	{
	case 1: currentScene->Destroy();	//clean up before removing it
			delete currentScene;
			currentScene = new SceneTest(); //change the pointer
			currentScene->Reset();			//set the camera up correct
			break;
	case 2: currentScene->Destroy();
			delete currentScene;
			currentScene = new SceneTest2();
			currentScene->Reset();
			break;
	case 3: currentScene->Destroy();
			delete currentScene;
			currentScene = new SceneTest3();
			currentScene->Reset();
			break;
```

```
        case 4: currentScene->Destroy();
                        delete currentScene;
                        currentScene = new SceneTest4();
                        currentScene->Reset();
                        break;
        }
        timer.Reset();
}

void init()
{
        setupScenes();
        fogSelect = true;
        lightSelect = true;
        freeCamera = true;
        paused = false;

        hasOpenedFile = false;

        timer.Reset();
}

void openFile()
{
        if(currentScene->getID() == 3)
        {
                int curTime = timer.GetTime();
                outputFile = fopen ("outputPCSS.txt","w");
                fprintf(outputFile,"Num Cubes, FPS \n",curTime);
        }
        else if(currentScene->getID() == 4)
        {
                outputFile = fopen ("outputBMSS.txt","w");
        }
        else
        {
                outputFile = fopen ("Othertext.txt","w");
        }
}

void outputText()
{
        if(!hasOpenedFile)
        {
                openFile();
                hasOpenedFile = true;
        }

        fpsCounter.Update();
        currentScene->getInfo(outputFile);
        fprintf(outputFile," %f \n",fpsCounter.GetFps());

}

void closeFile()
{
        int curTime = (int) timer.GetTime();
        //fprintf(outputFile,"End log at %d \n",curTime);
        fclose (outputFile);
        hasOpenedFile = false;
}
```

```
void displayFramerate()          //this is the text that gets displayed on the screen
{
        glDisable(GL_LIGHTING);
        fpsCounter.Update();

        //Print fps
        static char fpsString[32];
        sprintf(fpsString, "%.2f", fpsCounter.GetFps());

        //Set matrices for ortho
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(-1.0f, 1.0f, -1.0f, 1.0f);

        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();

        glColor3f(1.0f, 0.0f, 0.0f);

        //Print text
        glRasterPos2f(-1.0f, 0.9f);
        for(unsigned int i=0; i<strlen(fpsString); ++i)
                glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, fpsString[i]);

        //reset matrices
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();
        glEnable(GL_LIGHTING);
}

static void displayGL(void)
{
        float currentTime = timer.GetTime();
        //checkTime(currentTime);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        currentScene->Render(currentTime);
        outputText();
        displayFramerate();
        glutSwapBuffers();
        glFinish();
}

static void reshapeGL(int w, int h)
{
        if (h == 0)
                h = 1;
        float ratio =  w * 1.0 / h;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glViewport(0, 0, w, h);
        gluPerspective(45,ratio,1,10000);
        glMatrixMode(GL_MODELVIEW);
}

static void quit(void)
{
        //closeFile();
        std::cout << "Exiting..." << std::endl;
        exit(0);
}


static void initGL(int argc, char **argv)
{
        glEnable(GL_CULL_FACE);
        glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
        glClearColor(0.0f,0.0f,0.0f,1.0f);
        glEnable(GL_DEPTH_TEST);
}
```

```
static void glutKeyboardFunc(unsigned char key, int x, int y)
{
switch (key){
        case 27 :                       quit();
                        break;
        case 'w' :              currentScene->MoveForward();
                        break;
        case 'a' :              currentScene->StrafeLeft();
                        break;
        case 's' :              currentScene->MoveBackward();
                        break;
        case 'd' :              currentScene->StrafeRight();
                        break;

        case 'c':                       currentScene->camera->setFreelook(freeCamera);
                                        freeCamera = freeCamera ? false : true;
                        break;

        case 'l':                       currentScene->switchCamera();
                        break;

        case 'p': if(paused)
                          {
                                  timer.Unpause();
                          }
                          else
                          {
                                  timer.Pause();
                          }
                          paused = paused ?  false  : true;
                          break;
        case ' ': currentScene->SpaceBar();
                        break;

//SCENE CHANGERS
        case '1':       changeScene(1);
                        break;
        case '2':       changeScene(2);
                        break;
        case '3':       changeScene(3);
                        break;
        case '4':       changeScene(4);
                        break;
        }
}
static void glutPassiveMotionFunc (int x, int y)         {currentScene->camera->mouseMove(x , y);}
int main(int argc, char** argv)
{
        int winPosX = 800;
        int winPosY = 150;
        winWidth = 800;
        winHeight = 600;
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );

        glutInitWindowPosition(winPosX, winPosY);
        glutInitWindowSize(winWidth,winHeight);

        bool mainWindow = glutCreateWindow("Shadows");
        if(!mainWindow){ return 1;}

        glutReshapeFunc(reshapeGL);
        glutDisplayFunc(displayGL);
        glutIdleFunc(displayGL);
        glutKeyboardFunc(glutKeyboardFunc);
        glutPassiveMotionFunc(glutPassiveMotionFunc);

        printf(" --OpenGL Version: \t %s --\n\n",glGetString(GL_VERSION));

  initGL(argc, argv);
  init();
  glutMainLoop();
  return 0;
}
```

Shader.cpp

```
#include "Shader.h"
#include <string.h>
#include <iostream>

using namespace std;

static char* textFileRead(const char *fileName)
{
	char* text;

	if (fileName != NULL)
	{
	FILE *file = fopen(fileName, "rt");

		if (file != NULL)
		{
	    fseek(file, 0, SEEK_END);
	    int count = ftell(file);
	    rewind(file);

			if (count > 0)
			{
				text = (char*)malloc(sizeof(char) * (count + 1));
				count = fread(text, sizeof(char), count, file);
				text[count] = '\0';
			}
			fclose(file);
		}
	}
	return text;
}

static void validateShader(GLuint shader, const char* file = 0)
{
	const unsigned int BUFFER_SIZE = 512;
	char buffer[BUFFER_SIZE];
	memset(buffer, 0, BUFFER_SIZE);
	GLsizei length = 0;

	glGetShaderInfoLog(shader, BUFFER_SIZE, &length, buffer);
	if (length > 0)
	{
		cout << "Shader " << shader << " (" << (file?file:"") << "): " << buffer << endl;
	}
}

static void validateProgram(GLuint program)
{
	const unsigned int BUFFER_SIZE = 512;
	char buffer[BUFFER_SIZE];
	memset(buffer, 0, BUFFER_SIZE);
	GLsizei length = 0;

	memset(buffer, 0, BUFFER_SIZE);
	glGetProgramInfoLog(program, BUFFER_SIZE, &length, buffer);
	if (length > 0)
	{
		cout << "Program " << program << ": " << buffer << endl;
	}

	glValidateProgram(program);
	GLint status;
	glGetProgramiv(program, GL_VALIDATE_STATUS, &status);
	if (status == GL_FALSE)
	{
		cout << "Error validating shader " << program << endl;
	}
}
```

```
Shader::Shader(const char *vsFile, const char *fsFile)
{
    init(vsFile, fsFile);
}

void Shader::init(const char *vsFile, const char *fsFile)
{
        shader_vp = glCreateShader(GL_VERTEX_SHADER);
        shader_fp = glCreateShader(GL_FRAGMENT_SHADER);

        const char* vsText = textFileRead(vsFile);
        const char* fsText = textFileRead(fsFile);

    if (vsText == NULL || fsText == NULL)
        {
        cerr << "Either vertex shader or fragment shader file not found." << endl;
        return;
    }

        glShaderSource(shader_vp, 1, &vsText, 0);
        glShaderSource(shader_fp, 1, &fsText, 0);

        glCompileShader(shader_vp);
        validateShader(shader_vp, vsFile);
        glCompileShader(shader_fp);
        validateShader(shader_fp, fsFile);

        shader_id = glCreateProgram();
        glAttachShader(shader_id, shader_fp);
        glAttachShader(shader_id, shader_vp);
        glLinkProgram(shader_id);
        validateProgram(shader_id);

        printf("Created vertex shader: %s and fragment shader: %s \n",vsFile,fsFile);
}

Shader::~Shader()
{
        glDetachShader(shader_id, shader_fp);
        glDetachShader(shader_id, shader_vp);

        glDeleteShader(shader_fp);
        glDeleteShader(shader_vp);
        glDeleteProgram(shader_id);
}

void Shader::setUniformf(const std::string& sName, float fValue)
{
        /// Need to Activate the shader before, but we are not doing it
        /// in here to save computation time (if we are setting several variable at the same time)
        glUniform1f(glGetUniformLocation(shader_id, sName.c_str()), fValue);
}

void Shader::setUniformi(const std::string& sName, int fValue)
{
        /// Need to Activate the shader before, but we are not doing it
        /// in here to save computation time (if we are setting several variable at the same time)
        glUniform1i(glGetUniformLocation(shader_id, sName.c_str()), fValue);
}

void Shader::setUniformTexture(const std::string& sName, int iUnit)
{
        glUniform1iARB(glGetUniformLocationARB(shader_id, sName.c_str()), iUnit);
}

void Shader::setUniform3f(const std::string& sName, Vec3 values)
{
        glUniform3fARB(glGetUniformLocationARB(shader_id, sName.c_str()),values.x,values.y,values.z);
}

void Shader::setUniformTexture(const GLuint location, int iUnit)
{
        glUniform1iARB(location,iUnit);
}
```

```
void Shader::setUniformMatrix4fv(const std::string& sName, float * pValue)
{
	glUniformMatrix4fvARB(glGetUniformLocationARB(shader_id,sName.c_str()),1,GL_FALSE,pValue);
}

GLhandleARB Shader::id() {
	return shader_id;
}

GLuint Shader::getUniform(const std::string& sName)
{
	return glGetUniformLocationARB(shader_id,sName.c_str());
}

void Shader::bind() {
	glUseProgram(shader_id);
}

void Shader::unbind() {
	glUseProgramObjectARB(0);
}
```

FBO.cpp

```
#include "FBO.h"

#include <iostream>
#include <assert.h>

FBO::FBO(unsigned int iWidth, unsigned int iHeight):
m_iWidth(iWidth),
m_iHeight(iHeight)
{
	printf("Created an FBO! \n");

	glGenTextures(1, &texDepthID);
	glBindTexture(GL_TEXTURE_2D, texDepthID);

	printf("Generating Depth only \n");

	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);

	//we create a special texture for depth
	glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, m_iWidth, m_iHeight, 0,
		GL_DEPTH_COMPONENT, GL_FLOAT, 0);
	glBindTexture(GL_TEXTURE_2D, 0);

	//Generating ID
	glGenFramebuffersEXT(1, &m_iId);
	activate();

	//We attach the texture to the depth attachment point
	glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,GL_DEPTH_ATTACHMENT_EXT,GL_TEXTURE_2D,
	texDepthID, 0);

	//In order to avoid Color in the depth buffer
	glDrawBuffer(GL_NONE);
	glReadBuffer(GL_NONE);

	deactivate();

	//Check FBO status
	if(!checkFramebufferStatus())
		std::cerr<<"ERROR : FBO creation Fail "<<std::endl;

	//Depth Only
}
```

```
void FBO::enableDepthTexture()
{
	glEnable(GL_TEXTURE_2D);
	glBindTexture(GL_TEXTURE_2D,texDepthID);
}

void FBO::disableDepthTexture()
{
	glBindTexture(GL_TEXTURE_2D,0);
	glDisable(GL_TEXTURE_2D);
}

void FBO::activate()
{
	m_bIsActivated = true;
	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, m_iId);
}

void FBO::deactivate()
{
	m_bIsActivated = false;
	glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
}

void FBO::destroy()
{
	printf("Deleting the FBO \n\n\n");
	glDeleteFramebuffersEXT(1, &m_iId);
	glDeleteTextures(1,&m_iTexId);
	glDeleteRenderbuffersEXT(1, &m_iRenderId);
	m_bIsActivated = false;
}
```

SceneTest3.cpp

```
#include <Windows.h>
#include "SceneTest3.h"
#include "FBO.h"
#include <math.h>

float lightSphereSize = 0.2;	//the radius of the light sphere
int shadowMapQuality = 8;		//best if it's a multiple of 2
int numCubes = 125;

void SceneTest3::Init()
{
	glEnable(GL_LIGHTING);
	this->setID(3);
	viewFromLight = false;
	numberItems = 5;//this is the size of one side of the cube, so 5 = 5x5x5 cube
	//this is the only place the position and lookat values can be changed
	Vec3 lightPosition	= Vec3(10.0,	25.0,	-10.0);
	Vec3 lookingAt		= Vec3(0,	0,	0);
	light			= new Light(lightPosition, lookingAt, 0.5);
	float lightDiff[]	=  { 0.5f,0.5f,0.5f  };
	float lightAmb[]=  { 0.3f,0.3f,0.3f  };
	glMaterialfv(GL_FRONT, GL_AMBIENT, lightAmb);
	glMaterialfv(GL_FRONT, GL_DIFFUSE, lightDiff);
	glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

	//creating a new FBO for depth
	shadowMapFBO	= NULL;
	iShadowMapWidth  = shadowMapQuality*glutGet(GLUT_WINDOW_WIDTH);
	iShadowMapHeight= shadowMapQuality*glutGet(GLUT_WINDOW_HEIGHT);
	shadowMapFBO = new FBO(iShadowMapWidth,iShadowMapHeight);

	//Shader for working with depth values
	shader = new Shader("PCSS.vert","PCSS.frag");
	shadowUniform = shader->getUniform("ShadowMap");
	camera->setFreelook(false);
	camera->changePosition(Vec3(5.0f,10.0f,5.0f));
	camera->lookAt(Vec3(-5.0f, -5.0f, -5.0f));
}
```

```
void SceneTest3::renderLight()
{
        Vec3 lightPosition      = light->getPosition();
        Vec3 lightLookAt = light->getLookAt();

        glColor3f(1.0f, 1.0f, 1.0f);

        startTranslate(lightPosition.x,lightPosition.y,lightPosition.z);
        glutSolidSphere(lightSphereSize,20,20);
        endTranslate();

        glColor3f(1.0,1.0,0.0);
        glBegin(GL_LINES);
        glLineWidth(100.0f);
    glVertex3f(lightPosition.x,  lightPosition.y, lightPosition.z);
    glVertex3f(lightLookAt.x,    lightLookAt.y,           lightLookAt.z);
        glEnd();
}


void SceneTest3::getInfo(FILE* output)
{
        fprintf(output,"Numcubes: %d, ",numCubes);
}

void SceneTest3::RenderScene(int time)
{
        boolean cubes = false;

        glColor3f(0.5f, 0.5f, 0.5f);

        if(cubes)
        {
                numberItems = 125;

                int sizeLength = 1;

                int width  = sizeLength;
                int height = sizeLength;
                int depth  = sizeLength;

                numCubesScene4 = time/3000;
                if(numCubesScene4 > (sizeLength*sizeLength*sizeLength))
                {
                        numCubesScene4 = sizeLength*sizeLength*sizeLength;
                }

                for(int i = 0; i < numCubesScene4; ++i)
                {
                        int x = i % width;
                        int z = ( i / width ) % ( depth );
                        int y = i / ( width * depth);

                        float r = 0.8f + sinf((float)x)*0.5f;
                        float g = 0.8f + cosf((float)y)*0.5f;
                        float b = (r + g) * 0.5f;

                        glColor3f(r,g,b);
                        glPushMatrix();
                        startTranslate( x*2 - sizeLength , (y*2) +2 , z*2 - sizeLength);
                        glutSolidCube(0.8);
                        endTranslate();
                        glPopMatrix();
                }
        }
```

```
        else            //waffle
        {

                glTranslatef(0,5.0f,0);
                for(float x = -1.05; x<1.10; x+= 0.6)
                {
                        glBegin(GL_QUADS);
                                glVertex3f(x,           1.05,   0.0);
                                glVertex3f(x,           -1.05,  0.0);
                                glVertex3f(x+0.3,       -1.05,  0.0);
                                glVertex3f(x+0.3,       1.05,   0.0);
                        glEnd();
                }
                for(float x = -0.75; x<0.8; x+= 0.6)
                {
                        for(float y = 1.05; y > -1.10; y-= 0.6)
                        {
                                glBegin(GL_QUADS);
                                        glVertex3f( x,          y,              0.0);
                                        glVertex3f( x,          y-0.3,  0.0);
                                        glVertex3f( x+0.3,      y-0.3,  0.0);
                                        glVertex3f( x+0.3,      y,              0.0);
                                glEnd();
                        }
                }
                glTranslatef(0,-5,0);
        }
}

void SceneTest3::PreRender(int time)
{
        Vec3 lightPosition      = light->getPosition();
        Vec3 lightLookAt= light->getLookAt();

        shadowMapFBO->activate();

        shader->unbind();

        glViewport(0, 0, (GLint)iShadowMapWidth, (GLint)iShadowMapHeight);

        glClear(GL_DEPTH_BUFFER_BIT);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(75.0f, float(iShadowMapWidth)/float(iShadowMapHeight), 0.4, 10000.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        gluLookAt(      lightPosition.x,lightPosition.y,lightPosition.z,
                        lightLookAt.x,  lightLookAt.y,  lightLookAt.z,
                        0.0f,           1.0f,           0.0f);

        glCullFace(GL_BACK);
        RenderWalls();
        RenderScene(time);

        glGetFloatv(GL_MODELVIEW_MATRIX,lightViewMatrix.val1D);
        glGetFloatv(GL_PROJECTION_MATRIX,lightProjMatrix.val1D);
        light->setProjMatrix(lightProjMatrix);
        light->setViewMatrix(lightViewMatrix);

        shadowMapFBO->deactivate();
}

void SceneTest3::switchCamera()
{
        viewFromLight = viewFromLight == false ? true : false;
}
```

```
void SceneTest3::RenderCamera()
{
	if(viewFromLight)
	{
		Vec3 lightPosition	= light->getPosition();
		Vec3 lightLookAt = light->getLookAt();

		gluLookAt(	lightPosition.x, lightPosition.y, lightPosition.z,
				lightLookAt.x,		lightLookAt.y,
	lightLookAt.z,
				0.0f,				1.0f,
	0.0f);
	}
	else
	{
		camera->reshapeCamera();
	}
}

void SceneTest3::Render(int time)
{
	Tick(time);
	PreRender(time);

	glViewport(0, 0, glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));

	glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	shader->bind();
	shader->setUniformi("lightWidth",light->getWidth());

	glUniform1iARB(shadowUniform,7);
	glActiveTextureARB(GL_TEXTURE7);
	shadowMapFBO->enableDepthTexture();

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(75.0f, float(iShadowMapWidth)/float(iShadowMapHeight), 0.4, 10000.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	RenderCamera();
	renderLight();

	//For use in my own shader

	Mat4 cameraView;
	Mat4 cameraProj;
	glGetFloatv(GL_MODELVIEW_MATRIX,cameraView.val1D);
	glGetFloatv(GL_PROJECTION_MATRIX,cameraProj.val1D);
	camera->setViewMatrix(cameraView);
	camera->setProjMatrix(cameraProj);

	shader->setUniformMatrix4fv("cameraViewMatrix",camera->getViewMatrix().val1D);
	shader->setUniformMatrix4fv("cameraProjMatrix",camera->getProjMatrix().val1D);
	shader->setUniformMatrix4fv("lightViewMatrix",light->getViewMatrix().val1D);
	shader->setUniformMatrix4fv("lightProjMatrix",light->getProjMatrix().val1D);


	glCullFace(GL_BACK);
	RenderScene(time);
	RenderWalls();

	shader->unbind();

//	RenderShadowbuffer();
}
```

```
void SceneTest3::Reset()
{
	printf("%s \n","Resetting SceneTest");
	camera->changePosition(Vec3(-10.0f,20.0f,0.0f));
	camera->lookAt(Vec3(-30.0f,25.0f,-30.0f));
}

void SceneTest3::Destroy()
{
	printf("%s \n","Cleaning up SceneTest");

	delete light;
	delete shader;

	Scene::Destroy();
}

void SceneTest3::Tick(float elapsedTime)
{
	float radius = 15;

	Vec3 lightPosition	= light ->getPosition();
	Vec3 lightLookAt = light ->getLookAt();

	float rx = cos(elapsedTime/1000.0f)* radius;
	float rz = sin(elapsedTime/1000.0f)* radius;

	Vec3 newPosition = Vec3(rx, lightPosition.y ,rz);

	light->setPosition(newPosition);
	//light->setLookAt(newPosition);
}
```

SceneTest4.cpp

```
#include <Windows.h>
#include "SceneTest4.h"

#include "FBO.h"
#include <math.h>

int numCubesScene4 = 0;

void SceneTest4::Init()
{
	glEnable(GL_LIGHTING);
	this->setID(4);
	viewFromLight = false;
	numberItems = 1;

	//this is the only place the position and lookat values can be changed
	Vec3 lightPosition	= Vec3(10.0,	10.0,	-10.0);
	Vec3 lookingAt		= Vec3(0,	0,	0);
	int width		= 4;

	light			= new Light(lightPosition, lookingAt, width);

	//creating a new FBO for depth
	shadowMapFBO	= NULL;

	float shadowQuality = 3;

	iShadowMapWidth		= shadowQuality*glutGet(GLUT_WINDOW_WIDTH);
	iShadowMapHeight = shadowQuality*glutGet(GLUT_WINDOW_HEIGHT);

	printf("Created a shadow map of size (%d,%d) \n",iShadowMapHeight,iShadowMapWidth);
	shadowMapFBO = new FBO(iShadowMapWidth,iShadowMapHeight);

	//Shader for working with depth values
	shadow = new Shader("OBM.vert","OBM.frag");

	shadow->bind();
	shadowUniform = shadow->getUniform("ShadowMap");
```

```
        shadow->setUniformi("shadowMapDim",iShadowMapWidth);
        printf("%d\n",iShadowMapWidth);
        shadow->setUniformi("lightSize",light->getWidth());

        shadow->unbind();

        camera->setFreelook(false);
        camera->changePosition(Vec3(5.0f,10.0f,5.0f));
        camera->lookAt(Vec3(-5.0f, -5.0f, -5.0f));

        printf("Created test scene 2 \n");
}

void SceneTest4::renderLight()
{
        Vec3 lightPosition      = light->getPosition();
        Vec3 lightLookAt= light->getLookAt();

        glColor3f(1.0f, 1.0f, 1.0f);

        startTranslate(lightPosition.x,lightPosition.y,lightPosition.z);
        glutSolidSphere(0.2,20,20);
        endTranslate();

        glColor3f(1.0,1.0,0.0);
        glBegin(GL_LINES);
        glLineWidth(100.0f);
    glVertex3f(lightPosition.x,  lightPosition.y, lightPosition.z);
    glVertex3f(lightLookAt.x,    lightLookAt.y,           lightLookAt.z);
        glEnd();
}

void SceneTest4::getInfo(FILE* output)
{
        fprintf(output,"Numcubes: %d, ",numCubesScene4);
}

void SceneTest4::RenderScene(int time)
{
        boolean cubes = false;
        glColor3f(0.5f, 0.5f, 0.5f);
        if(cubes)
        {
                numberItems = 125;
                int sizeLength = 1;
                int width  = sizeLength;
                int height = sizeLength;
                int depth  = sizeLength;

                numCubesScene4 = time/3000;
                if(numCubesScene4 > (sizeLength*sizeLength*sizeLength))
                {
                        numCubesScene4 = sizeLength*sizeLength*sizeLength;
                }

                for(int i = 0; i < numCubesScene4; ++i)
                {
                        int x = i % width;
                        int z = ( i / width ) % ( depth );
                        int y = i / ( width * depth);

                        float r = 0.8f + sinf((float)x)*0.5f;
                        float g = 0.8f + cosf((float)y)*0.5f;
                        float b = (r + g) * 0.5f;

                        glColor3f(r,g,b);
                        glPushMatrix();
                        startTranslate( x*2 - sizeLength , (y*2) +2 , z*2 - sizeLength);
                        glutSolidCube(0.8);
                        endTranslate();
                        glPopMatrix();
                }
        }
```

```
	Else		//waffle
	{
		glTranslatef(0,5.0f,0);
		for(float x = -1.05; x<1.10; x+= 0.6)
		{
			glBegin(GL_QUADS);
				glVertex3f(x,		1.05,	0.0);
				glVertex3f(x,		-1.05,	0.0);
				glVertex3f(x+0.3,	-1.05,	0.0);
				glVertex3f(x+0.3,	1.05,	0.0);
			glEnd();
		}
		for(float x = -0.75; x<0.8; x+= 0.6)
		{
			for(float y = 1.05; y > -1.10; y-= 0.6)
			{
				glBegin(GL_QUADS);
					glVertex3f( x,		y,		0.0);
					glVertex3f( x,		y-0.3,	0.0);
					glVertex3f( x+0.3,	y-0.3,	0.0);
					glVertex3f( x+0.3,	y,		0.0);
				glEnd();
			}
		}
		glTranslatef(0,-5,0);
	}
}

void SceneTest4::PreRender(int time)
{
	Vec3 lightPosition	= light->getPosition();
	Vec3 lightLookAt= light->getLookAt();

	shadowMapFBO->activate();

	shadow->unbind();

	glViewport(0, 0, (GLint)iShadowMapWidth, (GLint)iShadowMapHeight);

	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

	float zNear = 1.0f;
	float zFar = 1000.0f;

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(60.0f, float(iShadowMapWidth)/float(iShadowMapHeight), zNear, zFar);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	gluLookAt(	lightPosition.x,	lightPosition.y,	lightPosition.z,
			lightLookAt.x,		lightLookAt.y,		lightLookAt.z,
			0.0f,			1.0f,			0.0f);

	glGetFloatv(GL_MODELVIEW_MATRIX,lightViewMatrix.val1D);
	glGetFloatv(GL_PROJECTION_MATRIX,lightProjMatrix.val1D);
	light->setProjMatrix(lightProjMatrix);
	light->setViewMatrix(lightViewMatrix);

	shadow->bind();
	shadow->setUniformf("zNear",zNear);
	shadow->setUniformf("zFar",zFar);
	shadow->unbind();

	glCullFace(GL_BACK);
	RenderWalls();
	RenderScene(time);

	shadowMapFBO->deactivate();
}
```

```
void SceneTest4::switchCamera()
{
	viewFromLight = viewFromLight == false ? true : false;
}

void SceneTest4::RenderCamera()
{
	if(viewFromLight)
	{
		Vec3 lightPosition	= light->getPosition();
		Vec3 lightLookAt = light->getLookAt();
		gluLookAt(	lightPosition.x,	lightPosition.y, lightPosition.z,
				lightLookAt.x,		lightLookAt.y,	lightLookAt.z,
				0.0f,			1.0f,		0.0f);
	}
	else
	{
		camera->reshapeCamera();
	}
}

void SceneTest4::Render(int time)
{
	Tick(time);
	PreRender(time);

	glViewport(0, 0, glutGet(GLUT_WINDOW_WIDTH), glutGet(GLUT_WINDOW_HEIGHT));

	glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	shadow->bind();

	glUniform1iARB(shadowUniform,7);
	glActiveTextureARB(GL_TEXTURE7);
	shadowMapFBO->enableDepthTexture();

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(75.0f, float(iShadowMapWidth)/float(iShadowMapHeight), 0.4, 1000.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	RenderCamera();
	renderLight();

	//For use in my own shadow

	Mat4 cameraView;
	Mat4 cameraProj;
	glGetFloatv(GL_MODELVIEW_MATRIX,cameraView.val1D);
	glGetFloatv(GL_PROJECTION_MATRIX,cameraProj.val1D);
	camera->setViewMatrix(cameraView);
	camera->setProjMatrix(cameraProj);

	shadow->setUniformMatrix4fv("cameraViewMatrix",camera->getViewMatrix().val1D);
	shadow->setUniformMatrix4fv("cameraProjMatrix",camera->getProjMatrix().val1D);
	shadow->setUniformMatrix4fv("lightViewMatrix",light->getViewMatrix().val1D);
	shadow->setUniformMatrix4fv("lightProjMatrix",light->getProjMatrix().val1D);

	glCullFace(GL_BACK);
	RenderScene(time);
	RenderWalls();

	shadow->unbind();
}

void SceneTest4::Reset()
{
	printf("%s \n","Resetting SceneTest");
	camera->changePosition(Vec3(-10.0f,20.0f,0.0f));
	camera->lookAt(Vec3(-30.0f,25.0f,-30.0f));
}
```

```
void SceneTest4::Destroy()
{
	printf("%s \n","Cleaning up SceneTest");
	delete light;
	delete shadow;
	Scene::Destroy();
}

void SceneTest4::Tick(float elapsedTime)
{
	float radius = 10;

	Vec3 lightPosition	= light ->getPosition();
	Vec3 lightLookAt = light ->getLookAt();

	float rx = cos(elapsedTime/1000.0f)* radius;
	float rz = sin(elapsedTime/1000.0f)* radius;

	Vec3 newPosition = Vec3(rx, lightPosition.y ,rz);

	light->setPosition(newPosition);
	//light->setLookAt(newPosition);
}
```

PCSS.vert

```
uniform mat4 lightViewMatrix;
uniform mat4 lightProjMatrix;
uniform mat4 cameraViewMatrix;
uniform mat4 cameraProjMatrix;

varying vec4 ShadowCoord;
varying vec4 Pos;

void main()
{
	// This is the position in -1,-1, to 1,1 for a normal render
	// from the POV of the camera
	vec4 pos = ftransform();

	// This is the position of the vertex as seen from the light source
	// in the screen of the light source (-1,-1) to (1,1)
	// This undoes whatver ftransform did
	// puts the point back into world space and then puts
	// it into the lights POV on the screen.
	ShadowCoord = lightProjMatrix  * lightViewMatrix *
		inverse(cameraViewMatrix) * inverse(cameraProjMatrix) * pos;

	Pos = gl_Vertex.xyzw;

	gl_Position = pos;
	gl_FrontColor = gl_Color;
}
```

PCSS.frag

```
uniform sampler2D ShadowMap;
varying vec4 ShadowCoord;
varying vec4 Pos;

uniform int lightWidth;

int BLOCKER_SAMPLES = 16;

float numBlockers = 0;
```

```
vec2 poissonDisk[16] =
{
        vec2( -0.94201624, -0.39906216 ),
        vec2( 0.94558609, -0.76890725 ),
        vec2( -0.094184101, -0.92938870 ),
        vec2( 0.34495938, 0.29387760 ),
        vec2( -0.91588581, 0.45771432 ),
        vec2( -0.81544232, -0.87912464 ),
        vec2( -0.38277543, 0.27676845 ),
        vec2( 0.97484398, 0.75648379 ),
        vec2( 0.44323325, -0.97511554 ),
        vec2( 0.53742981, -0.47373420 ),
        vec2( -0.26496911, -0.41893023 ),
        vec2( 0.79197514, 0.19090188 ),
        vec2( -0.24188840, 0.99706507 ),
        vec2( -0.81409955, 0.91437590 ),
        vec2( 0.19984126, 0.78641367 ),
        vec2( 0.14383161, -0.14100790 )
};

float penumbraSize(float zReceiver, float zBlocker) //Parallel plane estimation
{
        return (zReceiver - zBlocker) / zBlocker;
}

float lightDepthValue(vec4 coordinate)
{
        vec2 vlup = (coordinate.xy / coordinate.w);
        vlup = (1.0 + vlup)*0.5;
        float distanceFromLight = texture2D(ShadowMap,vlup).z;
        return distanceFromLight;
}

float findBlockers()
{
        float currentDepth = ShadowCoord.z/ShadowCoord.w;
        currentDepth =  (1.0 + currentDepth) * 0.5;

        float totalDepth = 0.0f;
        numBlockers = 0;

        for(int i = 0; i<BLOCKER_SAMPLES; i++)
        {
                vec2 newSearchValue = vec2(ShadowCoord.x, ShadowCoord.y) * poissonDisk[i];
                vec4 searchCoord = ShadowCoord + vec4(newSearchValue.x/4,newSearchValue.y/4,0,0);
                float sampleDepth = lightDepthValue(searchCoord);

                if(sampleDepth < currentDepth && ShadowCoord.w > 0.0)
                {
                        totalDepth += sampleDepth;
                        numBlockers+=1;
                }
        }

        float averageDepth = 0.0;
        averageDepth = totalDepth/numBlockers+0.09;

        return averageDepth;
}

float shadowCalc(vec4 coordinate)
{
        float shadowCoordDist = coordinate.z / coordinate.w ;

        // Now we need this in 0 - 1
        shadowCoordDist = (1.0 + shadowCoordDist) * 0.5;

        // Eliminate z fighting
        shadowCoordDist -= 0.0009;

        // Squash to become a 2d point on the lights screen -1 to 1
        vec2 vlup = (coordinate.xy / coordinate.w);

        // We need this value in 0 -> 1 for the texture lookup
```

```
        vlup = (1.0 + vlup)*0.5;

        // Do the texture lookup
        float distanceFromLight = texture2D(ShadowMap,vlup).z;

        float shadow = 1.0;

        //      Ignore everything behind the light
        if (coordinate.w > 0.0)
        {
                shadow = distanceFromLight < shadowCoordDist ? 0.2 : 1.0 ;
        }
        return shadow;
}

float PCF_Filter(float filter, vec4 coordinate)
{
        float shadowFinal = 0.0f;
        for(float x = -filter; x < filter; x++)
        {
                for(float y = -filter ; y < filter ; y++)
                {
                        vec4 newCoord = coordinate + vec4(x/64,y/64,0,0);
                        shadowFinal += shadowCalc(newCoord);
                }
        }
        return shadowFinal /= ((filter*2)*(filter*2));
}

void main()
{
        float shadow = 0.0f;
        float avgBlockerDepth = 0.0f;
        numBlockers = 0;
        avgBlockerDepth = findBlockers();

        if(numBlockers < 1 )
        {
                shadow = 1.0f;
        }
        else
        {
                float distanceFromLight = lightDepthValue(ShadowCoord);
                float size = penumbraSize(distanceFromLight,avgBlockerDepth);
                size = (size + 10) / 10 ;
                shadow = PCF_Filter(10, ShadowCoord);
        }

        vec4 col = gl_Color * 0.8 + vec4(Pos.xyz,1.0) * 0.1 ;
        gl_FragColor =  shadow * col;
}
```

BPSS.vert

```
uniform mat4 lightViewMatrix;
uniform mat4 lightProjMatrix;
uniform mat4 cameraViewMatrix;
uniform mat4 cameraProjMatrix;

uniform int lightWidth; //assumes light is square

varying vec4 ShadowCoord;
varying vec4 ShadowCoordLinear;

varying vec4 Pos;

varying vec4 cameraPos;
```

```
void main()
{
	// This is the position in -1,-1, to 1,1 for a normal render
	// from the POV of the camera
	cameraPos = ftransform();

	// This is the position of the vertex as seen from the light source
	// in the screen of the light source (-1,-1) to (1,1)
	// This undoes whatver ftransform did
	// puts the point back into world space and then puts
	// it into the lights POV on the screen.

	ShadowCoord = lightProjMatrix  * lightViewMatrix *
		inverse(cameraViewMatrix) * inverse(cameraProjMatrix) * cameraPos;

	ShadowCoordLinear =   lightViewMatrix *
		inverse(cameraViewMatrix) * inverse(cameraProjMatrix) * cameraPos;

	Pos = gl_Vertex.xyzw;

	gl_Position = cameraPos;

	gl_FrontColor = gl_Color;
}
```

BPSS.frag

```
uniform sampler2D ShadowMap;

uniform int shadowResolution;
uniform int shadowMapDim;
uniform int lightSize;

uniform float zNear;
uniform float zFar;

varying vec4 ShadowCoord;
varying vec4 ShadowCoordLinear;
varying vec4 Pos;

varying vec4 cameraPos;

// Width of light in world space
float l = 20;

// Width of the light plane in world space
float w = 10.f;

// Depth of the light plane in world space
float n = 0.4f;

vec4 computeBackProjectedBounds(vec2 uvShadow, vec2 uvPoint, float zs, float z)
{
	// Width of shadow map in pixels
	float r = shadowMapDim;

	vec4 b;

	float du = uvShadow.x - uvPoint.x;
	float dv = uvShadow.y - uvPoint.y;

	b.x = du - 0.5;
	b.y = du + 0.5;
	b.z = dv - 0.5;
	b.w = dv + 0.5;

	float ws = zs * w/(n * r);

	float scale = ws * (z/(z-zs)) * 1/l;

	return b * scale;
}
```

```
float computeArea(vec4 bounds)
{
        vec4 clampedBounds = max(vec4(-0.5),min(bounds,vec4(0.5) ));
        return (clampedBounds.y - clampedBounds.x) + (clampedBounds.w - clampedBounds.z);
}

float computeKernelWidth(float z)
{
        // Width of shadow map in pixels
        float r = shadowMapDim;
        float v = r * l * (n/w) * (1/n - 1/z);
        return v;
}

void main()
{
        float visibility = 1.0f;

        float shadowCoordDist = ShadowCoord.z/ShadowCoord.w;

        shadowCoordDist = 0.5 + shadowCoordDist * 0.5;

        // Eliminate z fighting
        shadowCoordDist -= 0.0009;

        // Squash to become a 2d point on the lights screen -1 to 1
        vec2 vlup = (ShadowCoord.xy/ShadowCoord.w);

        // We need this value in 0 -> 1 for the texture lookup
        vlup = (1.0 + vlup)*0.5;

        // Do the texture lookup
        float distanceFromLight = texture2D(ShadowMap,vlup);

        //      Ignore everything behind the light
        if (ShadowCoord.w > 0.0)
        {
                float w = computeKernelWidth(shadowCoordDist);
                float hw = max(0,min(w,50.0f));
                // Do the back projection
                for(float i = -hw/2; i <= hw/2; ++i)
                {
                        for(float j = -hw/2; j <= hw/2; ++j)
                        {
                                vec2 offset = vec2(i,j)/shadowMapDim;
                                float v = texture2D(ShadowMap,vlup + offset);
                                if(v > shadowCoordDist)
                                {
                                        vec4 bounds = computeBackProjectedBounds( (vlup +
                                        offset)*(shadowMapDim-1),
                                        vlup * (shadowMapDim-1), v, shadowCoordDist );
                                        float area = computeArea(bounds) * 0.0010;
                                        visibility -= area;
                                }
                        }
                }
        }
        gl_FragColor = visibility * gl_Color;
}
```