计算机图形学课程大项目

3D Flappy Bird

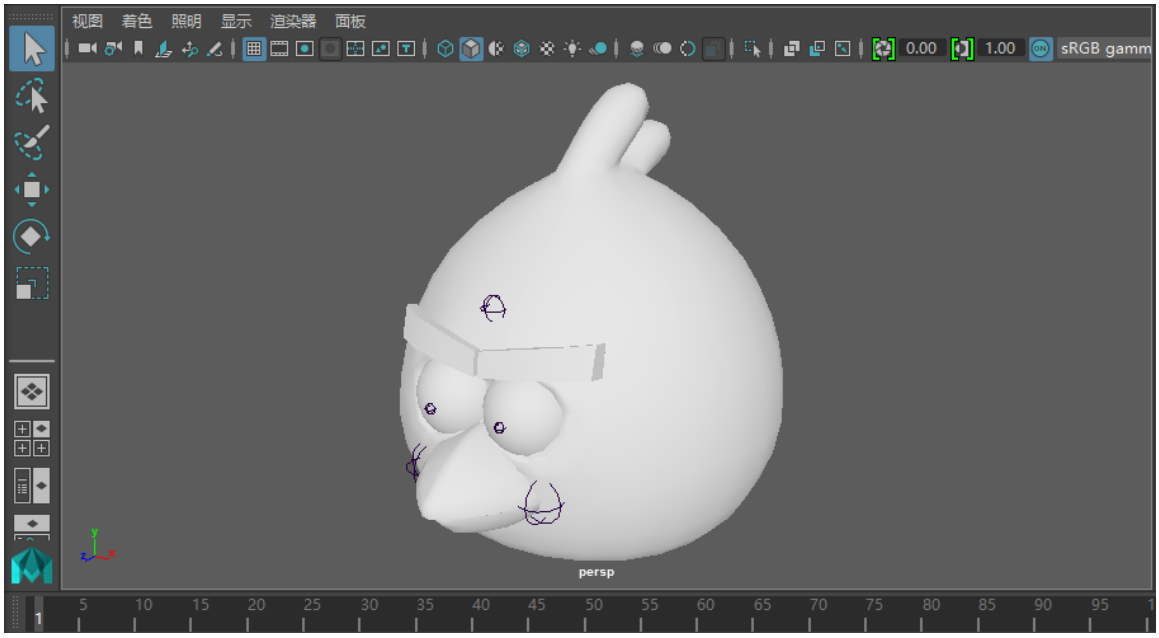
3140300075 刘曦
3140104579 王海容
3140300308 陈建宇

目录

Chapter 1: 建模表达基本体素

经过最后一次的图形学作业之后，我们学习到了如何初步的使用 Autodesk MAYA 建模工具来建模。最当初我们小组大程计划时我们想，若是一样透过 openGL 建模的话，我们能够设计的模型将会十分的单调。于是我们决定使用 MAYA 来为我们的游戏场景，主角，环境物体来建模。和助教确认过，我们是能够在网上找开源模型来使用的，于是我们在网上寻找了各种模型来尝试。起初我们对各种导入导出的模型文件格式还不是太熟悉，最开始很多模型都不具兼容性，所以我们在这方面花了许多时间去了解。

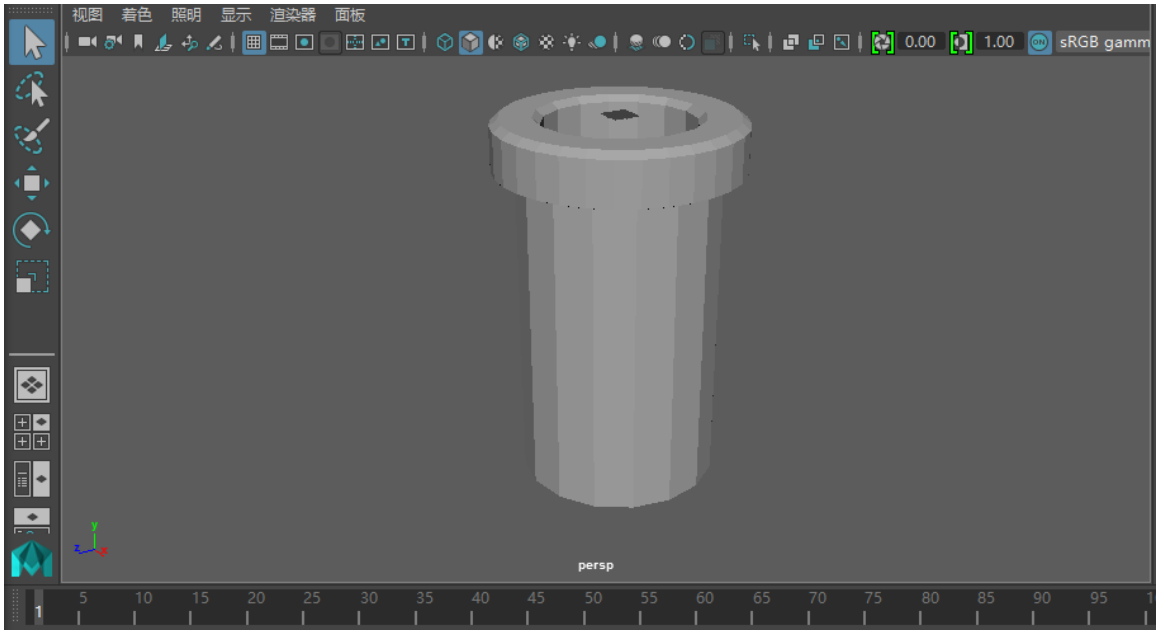
1.1 愤怒鸟的模型



愤怒鸟的模型从 The VG Resource – The Model Resource http://www.models-resource.com/mobile/angrybirdsgo/model/7029/ 网站下载下来之后，文件夹里头给了一个 fbx 文件以及 DAE 文件还有一个 bmp 纹理贴图如下

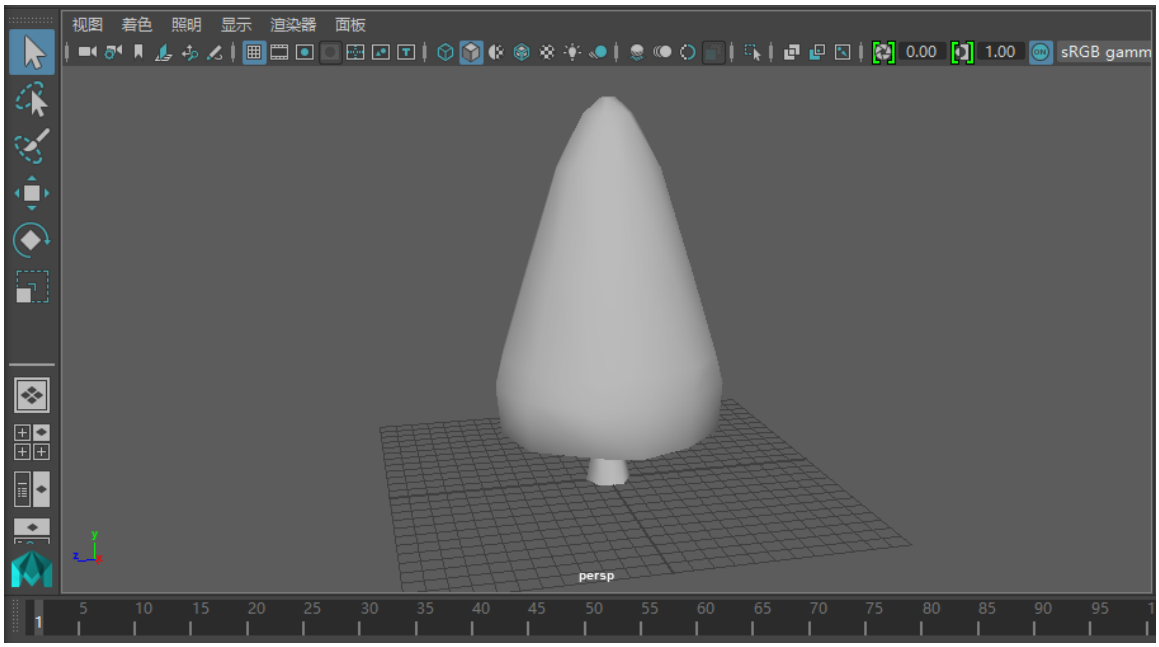
经过我们在网上了解到，fbx 是在各个平台都兼容的模型文件格式，于是我们就将此模型导入到 MAYA 里头，然后在把它导出成 obj 文件格式。最后在将这 obj 文件还有 bmp 纹理贴图交给 obj parser 来解析到 openGL 程序当中

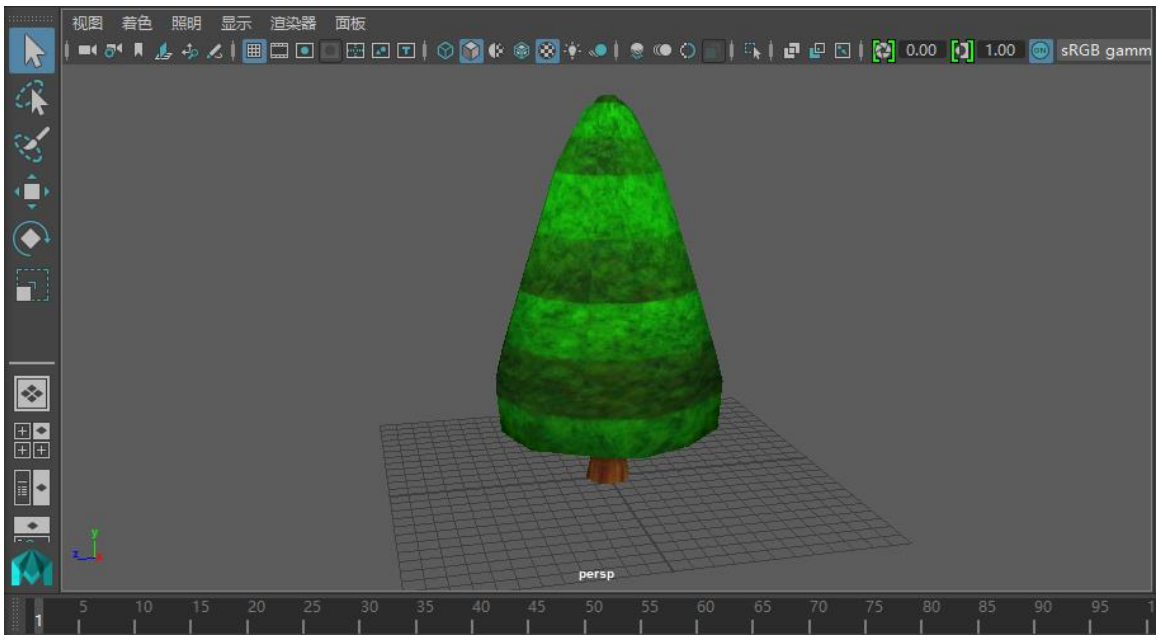
1.2 水管障碍物的模型



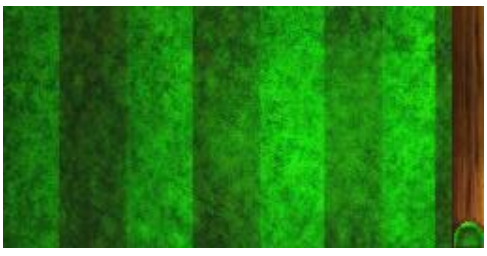
水管模型从网站下载下来之后，文件夹里头给了一个 3dx 文件以及 obj 文件。因为这个模型没有自带纹理贴图。所以我们将其 obj 解析进去 openGL 程序之后，在程序内赋予它材质。

1.3 树的模型





树的模型从 The VG Resource – The Model Resource 网站下载下来之后，文件夹里头给了一个 obj 文件以及 mtl 文件也就是附加于 obj 文件的材质文件，还有一个 bmp 纹理贴图如下



Chapter 2: OBJ 模型的导入与渲染

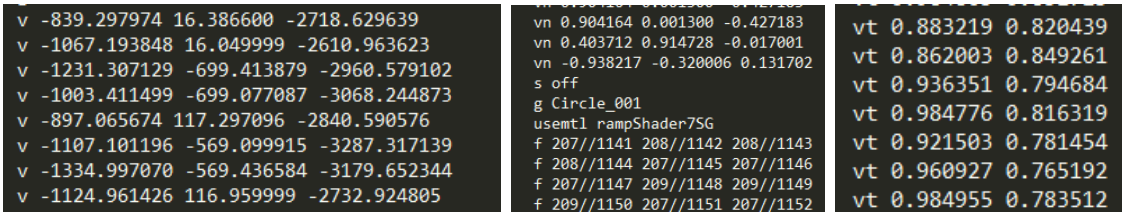
2.1 整体模型的导入与渲染

从 MAYA 生成.obj 格式的模型文件后，模型的导入交给了 obj3dmodel 类来负责导入和渲染。

```
class obj3dmodel {
protected:
	struct vertex {
		double x, y, z;
	};
	struct texel {
		double x, y;
	};
	struct face {
		unsigned int v1, v2, v3;
	};
	vector <vertex> vertices;
	vector <texel> texcoords;
	vector <vertex> normalVertices;
	vector <face> faces;
	vector <face> texfaces;
	vector <face> normalFaces;
public:
	obj3dmodel();
	~obj3dmodel();
	void read(const char* path);
	void render();
};
```

结构体 Vertex 代表着模型里的每个顶点的坐标，texel 代表了每个 Texture coordinate 的坐标，face 用于代表每个面所包含的顶点。

数据在.obj 文件里大概会以以下格式记载：



接下来的前三个容器 vertices、texcoords、normalVertices 分别用于装载顶点，纹理坐标和法向量坐标。

后三个容器 faces、texfaces 和 normalFaces 分别用于记录组成模型的每个块小面所需要用到的坐标、纹理坐标和法向量坐标。

以下函数 read(const char& path)用于读取位于路径 path 的.obj 模型文件。

```
void obj3dmodel::read(const char* path) {
	FILE *file = fopen(path, "r");
	char lineHeader[128];
	char mtlpath[128];
	char mtlname[128];
	while (1) {
		int res = fscanf(file, "%s", lineHeader);
		if (res == EOF) break;
		if (strcmp(lineHeader, "v") == 0) {
			vertex v;
			fscanf(file, "%lf %lf %lf", &v.x, &v.y, &v.z);
			vertices.push_back(v);
		}
		else if (strcmp(lineHeader, "vt") == 0) {
			texel t;
			fscanf(file, "%lf %lf", &t.x, &t.y);
			texcoords.push_back(t);
		}
		else if (strcmp(lineHeader, "vn") == 0) {
			vertex n;
			fscanf(file, "%lf %lf %lf", &n.x, &n.y, &n.z);
			normalVertices.push_back(n);
		}
		else if (strcmp(lineHeader, "f") == 0) {
			face f;
			face n;
			face t;
			fscanf(file, "%d/%d/%d %d/%d/%d %d/%d/%d", &f.v1, &t.v1,
&n.v1, &f.v2, &t.v2, &n.v2, &f.v3, &t.v3, &n.v3);
			f.v1--; f.v2--; f.v3--;
			t.v1--; t.v2--; t.v3--;
			n.v1--; n.v2--; n.v3--;
			faces.push_back(f);
			texfaces.push_back(t);
			normalFaces.push_back(n);
		}
	}
	fclose(file);
}
```

当读到'v'开头的数据时代表着该行记录着模型顶点的坐标，vt 和 vn 分别代表着纹理坐标和法向量坐标。'f'开头的数据则代表着模型的每个面所需要用到的坐标、纹理坐标和法向量坐标。

render()函数用于读取从前者 read()函数所读到的数据并进行渲染。

```
void obj3dmodel::render() {
	vertex v1, v2, v3, v4;
	vertex n1, n2, n3, n4;
	texel t1, t2, t3, t4;
	glBegin(GL_TRIANGLES);
	for (unsigned int i = 0; i < faces.size(); i++) {
		v1 = vertices[faces.at(i).v1];
		v2 = vertices[faces.at(i).v2];
		v3 = vertices[faces.at(i).v3];
		t1 = texcoords[texfaces.at(i).v1];
		t2 = texcoords[texfaces.at(i).v2];
		t3 = texcoords[texfaces.at(i).v3];
		n1 = normalVertices[normalFaces.at(i).v1];
		n2 = normalVertices[normalFaces.at(i).v2];
		n3 = normalVertices[normalFaces.at(i).v3];
		glNormal3f(n1.x, n1.y, n1.z);
		glTexCoord2f(t1.x, t1.y);
		glVertex3f(v1.x, v1.y, v1.z);
		glNormal3f(n2.x, n2.y, n2.z);
		glTexCoord2f(t2.x, t2.y);
		glVertex3f(v2.x, v2.y, v2.z);
		glNormal3f(n3.x, n3.y, n3.z);
		glTexCoord2f(t3.x, t3.y);
		glVertex3f(v3.x, v3.y, v3.z);
	}
	glEnd();
}
```

模型顶点 v1,v2,v3、纹理坐标 t1、t2、t3 和法向量坐标 n1、n2、n3 用于记录模型每一面所包含的坐标，纹理坐标和法向量坐标。然后在下列代码中用每一个顶点的 x、y、z 坐标进行渲染。

```
glBegin(GL_TRIANGLES);
for (unsigned int i = 0; i < faces.size(); i++) {
    v1 = vertices[faces.at(i).v1];
    v2 = vertices[faces.at(i).v2];
    v3 = vertices[faces.at(i).v3];
    t1 = texcoords[texfaces.at(i).v1];
    t2 = texcoords[texfaces.at(i).v2];
    t3 = texcoords[texfaces.at(i).v3];
    n1 = normalVertices[normalFaces.at(i).v1];
    n2 = normalVertices[normalFaces.at(i).v2];
    n3 = normalVertices[normalFaces.at(i).v3];
    glNormal3f(n1.x, n1.y, n1.z);
    glTexCoord2f(t1.x, t1.y);
    glVertex3f(v1.x, v1.y, v1.z);
    glNormal3f(n2.x, n2.y, n2.z);
    glTexCoord2f(t2.x, t2.y);
    glVertex3f(v2.x, v2.y, v2.z);
    glNormal3f(n3.x, n3.y, n3.z);
    glTexCoord2f(t3.x, t3.y);
    glVertex3f(v3.x, v3.y, v3.z);
}
```

以下 3 个皆是继承了 obj3dmodel 类的类，在其构造函数里调用了 obj3dmodel 的 read()函数进行模型读取，每个类里都有自己的 render()函数，在调用自己的 render()时都会开启纹理，进行对模型的转移和大小调节，最后调用父类 obj3dmodel 的 render()函数进行渲染。

```
class tube : public obj3dmodel
class tree : public obj3dmodel
class bird: public obj3dmodel
```

2.2 树(tree 类)模型渲染

tree 类的渲染函数：

```
void tree::render(GLuint treetexture)
{
	glPushMatrix();
	glTranslatef(x, y, z);
	glEnable(GL_TEXTURE_2D);
	glBindTexture(GL_TEXTURE_2D, treetexture);
	glRotatef(90, 1, 0, 0);
	switch (state) {
	case 0:
		glScalef(0.1f, 0.1f, 0.1f); break;
	case 1:
		glScalef(0.5f, 0.5f, 0.5f); break;
	case 2:
		glScalef(0.5f, 0.5f, 0.5f); break;
	}
	obj3dmodel::render();
	glDisable(GL_TEXTURE_2D);
	glPopMatrix();
}
```

tree 类的 state 变量会被随机分配一个 0~2 范围的值，当 state 为 0 时 tree 的父类函数 obj3drender::read()会读到 tree01.obj，为 2 时会读到 tree02.obj，为 3 时会读到房子模型以达到随机选择渲染模型的效果。房子渲染出来的效果并没有预期中理想，因此选择性放弃了房子的模型。因为每个模型的大小都不一样，因此在 render 函数里需要根据 state 的值来对模型的大小进行不同程度上的调整。

2.3 鸟(bird 类)模型渲染

bird 类的渲染函数：

```
void bird::render(GLuint birdtexture) {
       glEnable(GL_TEXTURE_2D);
       Glfloat birdcolor[] = { 0.309804, 0.184314, 0.184314 };
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, birdcolor);
       glPushMatrix();
       glScalef(0.01, 0.01, 0.01);
       glBindTexture(GL_TEXTURE_2D, birdtexture);
       glRotatef(90, 1.0, 0, 0.0);
       glRotatef(165, 0.0, 1.0, 0.0);
       glTranslatef(x, y, z);
    glRotatef(180*atan2f(lz,ly)/3.14159, -1, 0, 0);
       obj3dmodel::render();
       glPopMatrix();
       glDisable(GL_TEXTURE_2D);
}
```

2.4 水管(tube 类)模型渲染

水管类的渲染函数：

```
void tube::render(Gluint pipetexture) {
       Glfloat pipecolor[] = { 0.0f, 0.5f, 0.0f };
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, pipecolor);
       obj3dmodel::render();
}
```

因为水管没有适合的纹理图片，因此这里选择了直接调节材质而没有选择纹理映射。

Chapter 3: 材质与纹理的显示

本游戏带有材质的物体只有小鸟和水管。两者用到的材质都是对环境光和散射光的反应（GL_AMBIENT_AND_DIFFUSE）。尽管小鸟启用了水管，可是因为体积过小以及开启了纹理映射的关系，材质效果基本看不出来。

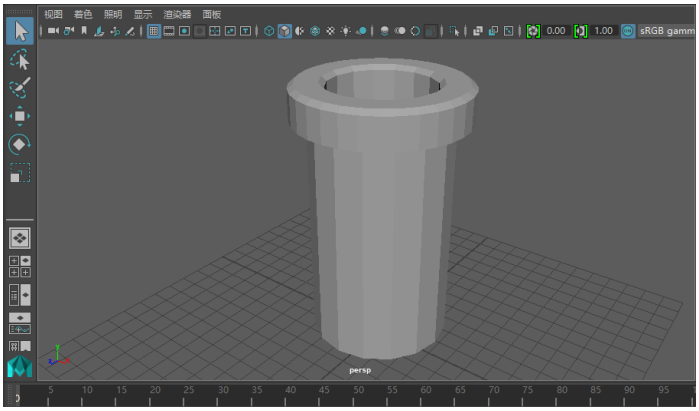
```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, pipecolor);
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, birdcolor);
```

开启材质：

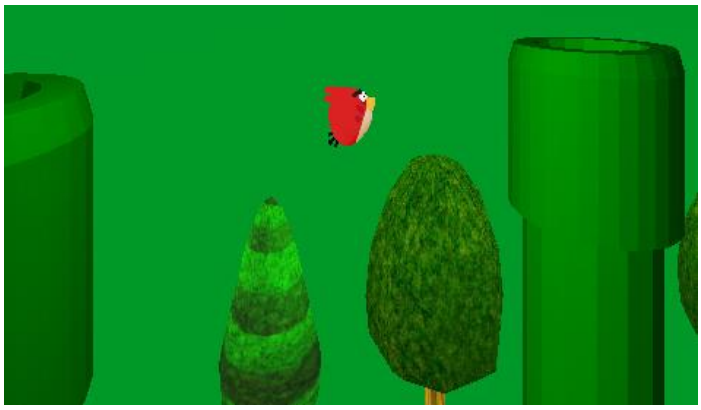
关闭材质：

水管因为没有才用到纹理映射，因此其材质效果看起来要明显很多。

材质开启前在 MAYA 导入的效果：



导入进游戏后经过 resize 和开启材质后的效果图：



纹理方面的 BMP 读取器 reference 了网上找来的 BMP loader.

引用：http://blog.csdn.net/hippig/article/details/7764990

其中用到的代码有：

```
void grab(void)
{
        FILE*     pDummyFile;
        FILE*     pWritingFile;
        Glubyte* pPixelData;
        Glubyte   BMP_Header[BMP_Header_Length];
        Glint     I, j;
        Glint     PixelDataLength;

        // 计算像素数据的实际长度
        I = WindowWidth * 3;    // 得到每一行的像素数据长度
        while (I % 4 != 0)       // 补充数据，直到i是的倍数
                ++I;                 // 本来还有更快的算法，
                                                        // 但这里仅追求直观，对速度没有太高要求
        PixelDataLength = I * WindowHeight;

        // 分配内存和打开文件
        pPixelData = (Glubyte*)malloc(PixelDataLength);
        if (pPixelData == 0)
                exit(0);

        pDummyFile = fopen("4__.bmp", "rb");
        if (pDummyFile == 0)
                exit(0);

        pWritingFile = fopen("4__.bmp", "wb");
        if (pWritingFile == 0)
                exit(0);

        // 读取像素
        glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
        glReadPixels(0, 0, WindowWidth, WindowHeight,
                GL_BGR_EXT, GL_UNSIGNED_BYTE, pPixelData);

        // 把dummy.bmp的文件头复制为新文件的文件头
        fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);
        fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
        fseek(pWritingFile, 0x0012, SEEK_SET);
        I = WindowWidth;
        j = WindowHeight;
        fwrite(&I, sizeof(i), 1, pWritingFile);
        fwrite(&j, sizeof(j), 1, pWritingFile);

        // 写入像素数据
        fseek(pWritingFile, 0, SEEK_END);
        fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

        // 释放内存和关闭文件
        fclose(pDummyFile);
        fclose(pWritingFile);
        free(pPixelData);
}
int power_of_two(int n)
{
        if (n <= 0)
                return 0;
        return (n & (n - 1)) == 0;
}
Gluint load_texture(const char* file_name)
```

```
{
	Glint width, height, total_bytes;
	Glubyte* pixels = 0;
	Glint last_texture_ID;
	Gluint texture_ID = 0;

	// 打开文件，如果失败，返回
	FILE* pFile = fopen(file_name, “rb”);
	if (pFile == 0)
		return 0;

	// 读取文件中图象的宽度和高度
	fseek(pFile, 0x0012, SEEK_SET);
	fread(&width, 4, 1, pFile);
	fread(&height, 4, 1, pFile);
	fseek(pFile, BMP_Header_Length, SEEK_SET);

	// 计算每行像素所占字节数，并根据此数据计算总像素字节数
	{
		Glint line_bytes = width * 3;
		while (line_bytes % 4 != 0)
			++line_bytes;
		total_bytes = line_bytes * height;
	}

	// 根据总像素字节数分配内存
	pixels = (Glubyte*)malloc(total_bytes);
	if (pixels == 0)
	{
		fclose(pFile);
		return 0;
	}

	// 读取像素数据
	if (fread(pixels, total_bytes, 1, pFile) <= 0)
	{
		free(pixels);
		fclose(pFile);
		return 0;
	}

	// 在旧版本的OpenGL中
	// 如果图象的宽度和高度不是的整数次方，则需要进行缩放
	// 这里并没有检查OpenGL版本，出于对版本兼容性的考虑，按旧版本处理
	// 另外，无论是旧版本还是新版本，
	// 当图象的宽度和高度超过当前OpenGL实现所支持的最大值时，也要进行缩放
	{
		Glint max;
		glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max);
		if (!power_of_two(width)
			|| !power_of_two(height)
			|| width > max
			|| height > max)
		{
			const Glint new_width = 256;
			const Glint new_height = 256; // 规定缩放后新的大小为边长的正方形
			Glint new_line_bytes, new_total_bytes;
			Glubyte* new_pixels = 0;

			// 计算每行需要的字节数和总字节数
			new_line_bytes = new_width * 3;
			while (new_line_bytes % 4 != 0)
				++new_line_bytes;
			new_total_bytes = new_line_bytes * new_height;

			// 分配内存
			new_pixels = (Glubyte*)malloc(new_total_bytes);
```

```
                        if (new_pixels == 0)
                        {
                                free(pixels);
                                fclose(pFile);
                                return 0;
                        }

                        // 进行像素缩放
                        gluScaleImage(GL_RGB,
                                width, height, GL_UNSIGNED_BYTE, pixels,
                                new_width, new_height, GL_UNSIGNED_BYTE, new_pixels);

                        // 释放原来的像素数据，把pixels指向新的像素数据，并重新设置width和height
                        free(pixels);
                        pixels = new_pixels;
                        width = new_width;
                        height = new_height;
                }
        }

        // 分配一个新的纹理编号
        glGenTextures(1, &texture_ID);
        if (texture_ID == 0)
        {
                free(pixels);
                fclose(pFile);
                return 0;
        }

        // 绑定新的纹理，载入纹理并设置纹理参数
        // 在绑定前，先获得原来绑定的纹理编号，以便在最后进行恢复
        glGetIntegerv(GL_TEXTURE_BINDING_2D, &last_texture_ID);
        glBindTexture(GL_TEXTURE_2D, texture_ID);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
                GL_BGR_EXT, GL_UNSIGNED_BYTE, pixels);
        glBindTexture(GL_TEXTURE_2D, last_texture_ID);

        // 之前为pixels分配的内存可在使用glTexImage2D以后释放
        // 因为此时像素数据已经被OpenGL另行保存了一份（可能被保存到专门的图形硬件中）
        free(pixels);
        return texture_ID;
}
```

以上三个函数的主要功能：

grab:读取 BMP 图片

power_of_two: 确保图片的大小是 2 的次方的，若不是的话会进行调整。

load_texture: 调用 grab 和 power_of_two，随后初始纹理属性和分配一个 texture ID。

纹理载入完后在 main 里保存被分配到的 texture ID.

e.g.

```
birdtexture = load_texture(“abtexture.bmp”);
```

随后在渲染时开启 GL_TEXTURE_2D 然后启用被分配到的 texture ID.

e.g.

```
void bird::render(Gluint birdtexture) {
       glEnable(GL_TEXTURE_2D);
       Glfloat birdcolor[] = { 0.309804, 0.184314, 0.184314 };
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, birdcolor);
       glPushMatrix();
       glScalef(0.01, 0.01, 0.01);
       glBindTexture(GL_TEXTURE_2D, birdtexture);
       glRotatef(90, 1.0, 0, 0.0);
       glRotatef(165, 0.0, 1.0, 0.0);
       glTranslatef(x, y, z);
    glRotatef(180*atan2f(lz,ly)/3.14159, -1, 0, 0);
       obj3dmodel::render();
       glPopMatrix();
       glDisable(GL_TEXTURE_2D);
}
```

调用 glBindTexture 函数使用指定的 texture ID 然后进行渲染即可。渲染结束后要记得调用 glDisable(GL_TEXTURE_2D)关闭纹理功能，否则 OpenGL 的当前状态会一直停留在纹理启动状态，导致后面不需要纹理的物体渲染出来也会带有同样的纹理。

Chapter 4: 几何变换功能

4.1 模型的缩放

因为每一个模型在 MAYA 所显示的大小比例和导入到 OpenGL 里显示的大小比例是不一样的，因此每一个模型在渲染前都必须经过一定比例的缩小。

比如 bird 的大小是经过了 1000 倍的缩小：

```
void bird::render(Gluint birdtexture) {
       glEnable(GL_TEXTURE_2D);
       Glfloat birdcolor[] = { 0.309804, 0.184314, 0.184314 };
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, birdcolor);
       glPushMatrix();
       glScalef(0.01, 0.01, 0.01);
       glBindTexture(GL_TEXTURE_2D, birdtexture);
       glRotatef(90, 1.0, 0, 0.0);
       glRotatef(165, 0.0, 1.0, 0.0);
       glTranslatef(x, y, z);
    glRotatef(180*atan2f(lz,ly)/3.14159, -1, 0, 0);
       obj3dmodel::render();
       glPopMatrix();
       glDisable(GL_TEXTURE_2D);
}
```

4.2 鸟模型的渲染

鸟飞行过程中，上升时头部需要向上旋转，下降时头不需要向下旋转。

```
glPushMatrix();
glScalef(0.01, 0.01, 0.01);
glBindTexture(GL_TEXTURE_2D, birdtexture);
glRotatef(90, 1.0, 0, 0.0);
glRotatef(165, 0.0, 1.0, 0.0);
glTranslatef(x, y, z);
glRotatef(180*atan2f(lz,ly)/3.14159, -1, 0, 0);
obj3dmodel::render();
glPopMatrix();
```

4.3 水管模型的渲染

水管的生成后默认在原点且朝向是 y 轴正方向，需要位移加旋转至所需位置

```
for (int i = 0; i < tubeAmount; i++)
 {
                        currentx = ttt[i].getx(); currenty = ttt[i].gety(); currentz =
tube_bottom;
                        glPushMatrix();
                        glTranslatef(currentx, currenty, currentz - 1.7);
                        glRotatef(90, 1.0, 0, 0.0);
                        ttt[i].render(pipetexture);
                        glPopMatrix();
 }
```

4.4 Buff 模型的渲染

buff 本身是个球体，在调用 gluSolidsphere 绘制球的的过程中需要平移和旋转

```
glPushMatrix();
glTranslatef(x, y, z);
glutSolidSphere(radius, 50, 50);
glPopMatrix();
glFlush();
```

4.5 分数的渲染

因为字符渲染模块将字符渲染到 xy 平面上，游戏中需要展示在 xz 和 yz 平面，因此需要平移和旋转确定分数的新位置。

```
glPushMatrix();
glTranslatef(bbb.getx()+6 * (cam.lx - cam.px)+10, bbb.gety()+6*(cam.ly- cam.py),
bbb.getz());
glRotatef(90, 1, 0, 0);
glRotatef(270* (cam.lx - cam.px)/cam2bird, 0, 1, 0);
```

Chapter 5: 光照功能

光照方面启用了环境光，散射光和反射光。 以下为光源的各个参数和位置：

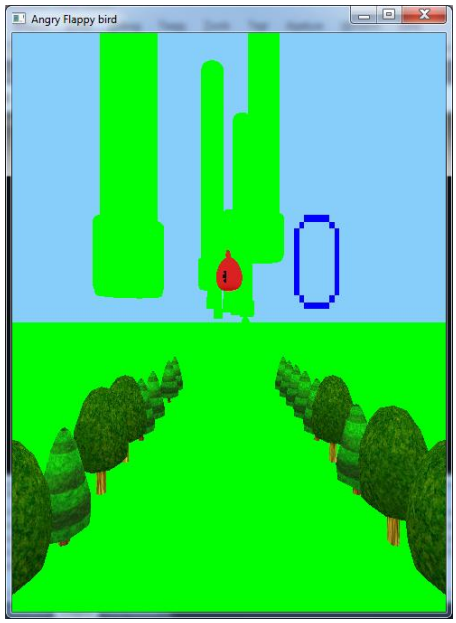
```
GLfloat ambient[] = { 0.2f, 0.2f, 0.2f };
GLfloat diffuse[] = { 0.8f, 0.8f, 0.8f };
GLfloat specular[] = { 0.0f, 0.0f, 0.0f };
GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f };
```

Position 的第四个参数被设置为了 1，因此光源的位置是固定的。本意是为了做到能从不同的位置看物体能看出不同的效果，不过可能因为启用了纹理映射的关系所以并看不出效果来。

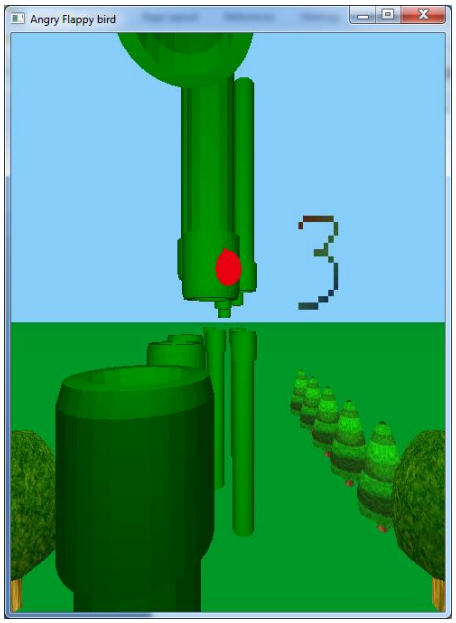
给光源定义完位置和光照参数后在 main 里分配给 GL_LIGHT0 并开启光照：

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
```

光照开启前：



光照开启后：



Chapter 6: 场景漫游

受游戏性的影响，本游戏的场景漫游仅局限于跟随鸟移动以及视角变换。
为了支持这些特性我们定义了摄像机类，其接口如下：

```
void idle(double x, double y);
void switchLeft();
void switchRight();
bool inRotation();
```

首先跟踪鸟的移动比较容易实现，由 idle 函数处理，idle 函数接受两个参数为鸟的平面坐标，因为摄像机高度固定，然后将 x，y 值传给 glulookat 函数作为 eyex，eyey 的参照坐标即可实现摄像机跟随鸟的移动。

视角变换通过 switchLeft 和 switchRight 确定，并由 idle 函数进行，分别对应两个方向的旋转 90 度。实质上只是做数学运算，确定旋转过程中 glulookat 函数参数的值。

计算过程如下：

```
timevar += DTIME;
px = x + DST*cos(timevar);
py = y - DST*sin(timevar);
if (px <= x + 0.1&&px >= x - 0.1)
{
	timevar = 0.0;
	stats = Y2B;
}
```

```
timevar += DTIME;
px = x + DST*sin(timevar);
py = y - DST*cos(timevar);
if (py <= y + 0.1&&py >= y - 0.1)
{
	timevar = 0.0;
	stats = X2B;
}
```

实时的旋转角度，由 timevar 确定。

px 为 eyex，py 为 eyey

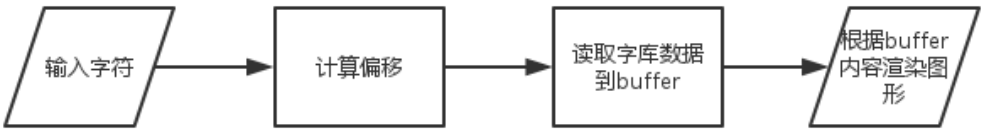
x 为 centerx，y 为 centery

inRotation 返回视角是否在变换，以辅助对鸟飞行的控制。

Chapter 7: Awsomeness 元素

7.1 字符立体展示

因为 opengl 不具有字符显示的功能，我们通过对字库的访问和 opengl 的渲染实现了字符在三维空间内的显示。实现流程图如下：



模块接口：

```
void textrender::drawText3(double x, double y, double z, double size, float r = 1.0,
float g = 1.0, float b = 1.0, int inverse = 0, int fade = 0, int kong = 0, int ilas =
0);
    void textrender::calOffset(const unsigned char word[3]);
    void textrender::calOffset(unsigned char a);
```

drawText3 函数实现对字符在三维空间内的渲染，参数分别对应渲染坐标（对应字符的左上角）、字符大小、颜色、是否倒置、是否有渐变效果、是否空心、是否斜体。

calOffset 函数重载以实现对中文字符和 ascii 表字符的字库偏移计算。

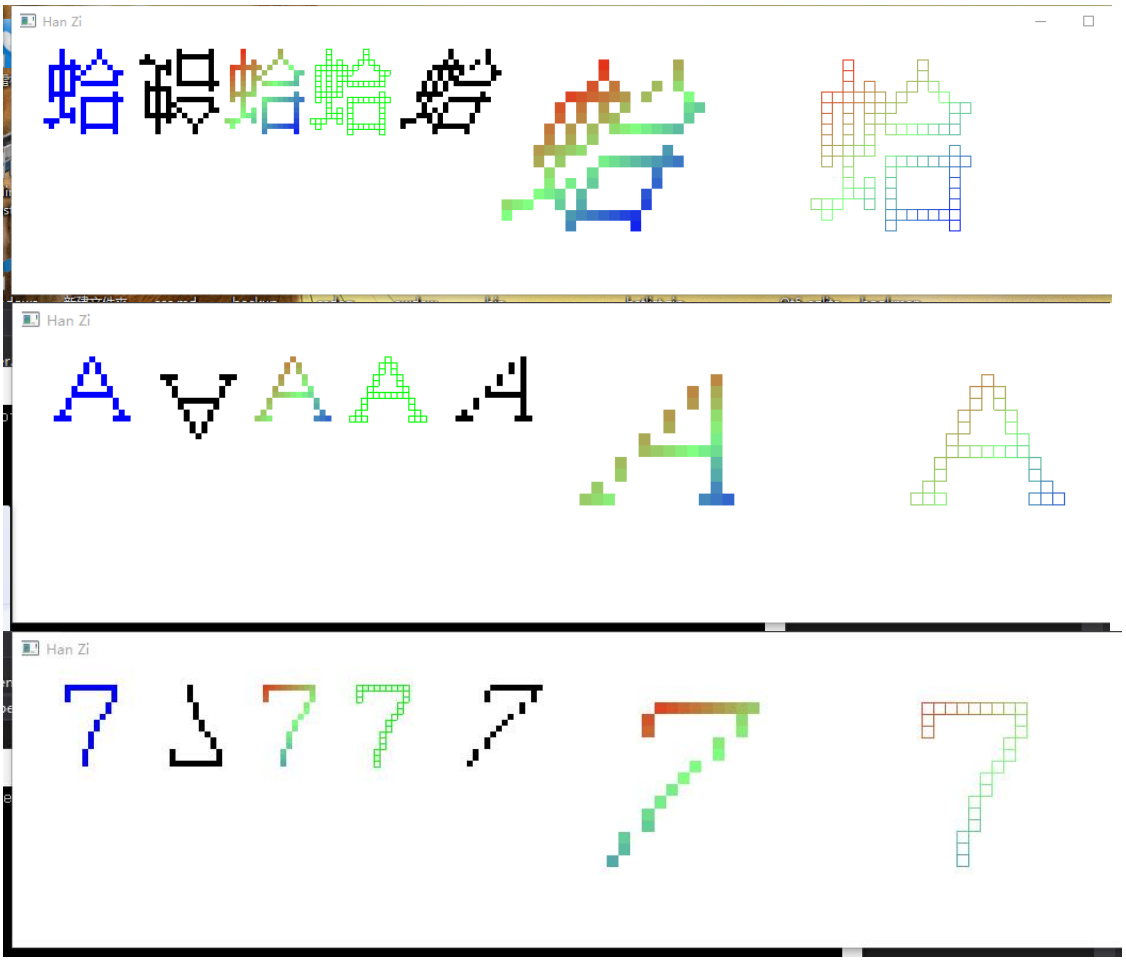
过程说明：

1) 偏移计算及字库内容读取
   所使用的字库为楼学庆计组课程中提供的字库 Hzk16，Hzk16 是符合 GB2312 标准的 16×16 点阵字库。

   计算公式如下：
   中文字符：
   offset = (94 * (unsigned int)(word[0] - 0xa0 - 1) + (word[1] - 0xa0 - 1)) * 32;

   英文及数字：
   offset= ((3 - 1) * 94 + a - 32 - 1) * 32;
2) 渲染图形
   根据 buffer 中的位数据，若为 1 绘制边长为 size 的矩形，这样倒置、渐变、斜体的实现也并不复杂、只是对矩形坐标和颜色的改变，不再赘述。而空心，则是只是渲染矩形前变更 glPolygonMode，也并不复杂。

渲染效果：



Chapter 8: Bonus 额外要求

8.1 碰撞检测

碰撞检测主要包括水管、地面与鸟的碰撞检测，buff 物品与鸟的碰撞检测。
地面的碰撞检测比较简单，比较地面的高度与鸟的高度即可。
水管的碰撞检测分为侧视视角主视视角两种情况。侧视视角下，为了模仿 2d 生物在 3d 空间内的生存状态，我们只对水管的侧面即 y 轴方向及顶部即 z 轴方向进行碰撞检测。代码如下：

```
if (X2B == rotk)
{
    if (zz < tube_bottom + h + bird_r&&yy>y - bird_r - tube_r&&yy < y + tube_r + bird_r
|| zz < tube_bottom)
    {
        ok = false;
     }
}
```

主视角下，因为考虑到水管的数量，同时进行三个方向上的碰撞检测，代码如下：

```
else
{
    if (zz < tube_bottom + h + bird_r&&yy>y - bird_r&&yy < y + tube_r + bird_r &&xx<x +
tube_r + bird_r&&xx>x - tube_r - bird_r || zz < tube_bottom)
    {
        ok = false;
    }
}
```

Buff 的碰撞检测为检测球心距离鸟的球心的距离是否小于两者的半径，代码如下：

```
double dxx = (x - this->x)*(x - this->x);
double dyy = (y - this->y)*(y - this->y);
double dzz = (z - this->z)*(y - this->z);
if (dxx + dyy + dzz < (radius + bird_r)*(radius + bird_r))
{
    activited = false;
    function = true;
               //…
}
```

Chapter 9: Reference 引用

愤怒鸟的模型 The VG Resource – The Model Resource
http://www.models-resource.com/mobile/angrybirdsgo/model/7029/

OpenGL 纹理入门
http://blog.csdn.net/hippig/article/details/7764990