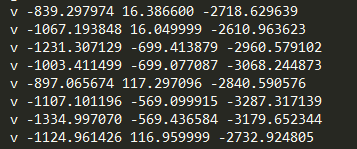
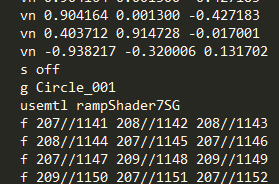
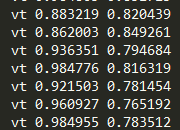
**2. OBJ模型的导入与渲染**

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。

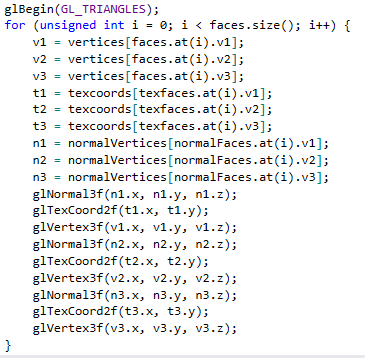
|  |
| --- |
| void obj3dmodel::read(const char\* path) {  FILE \*file = fopen(path, "r");  char lineHeader[128];  char mtlpath[128];  char mtlname[128];  while (1) {  int res = fscanf(file, "%s", lineHeader);  if (res == EOF) break;  if (strcmp(lineHeader, "v") == 0) {  vertex v;  fscanf(file, "%lf %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。



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 |

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。

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);  } |

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。

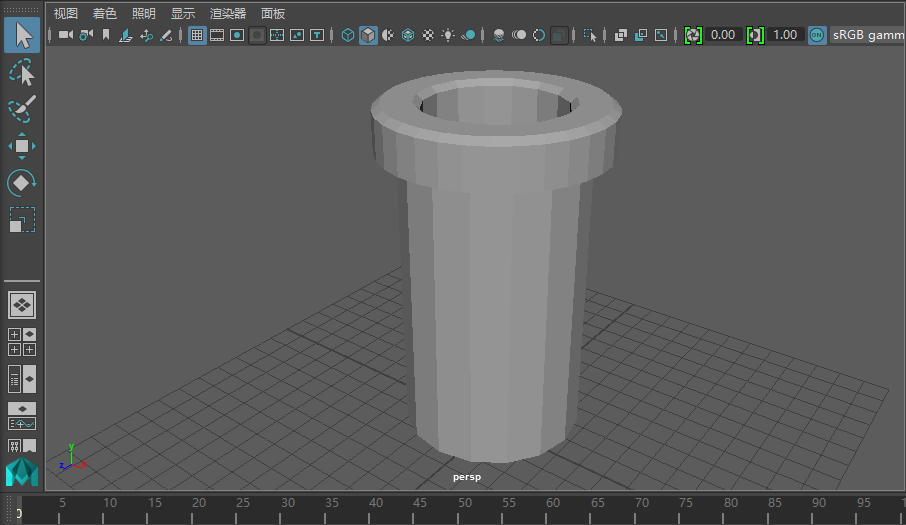
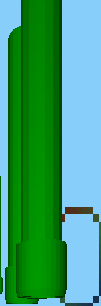
**3. 材质与纹理**

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。

**4.几何变换功能**

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);  } |

**5. 光照**

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：

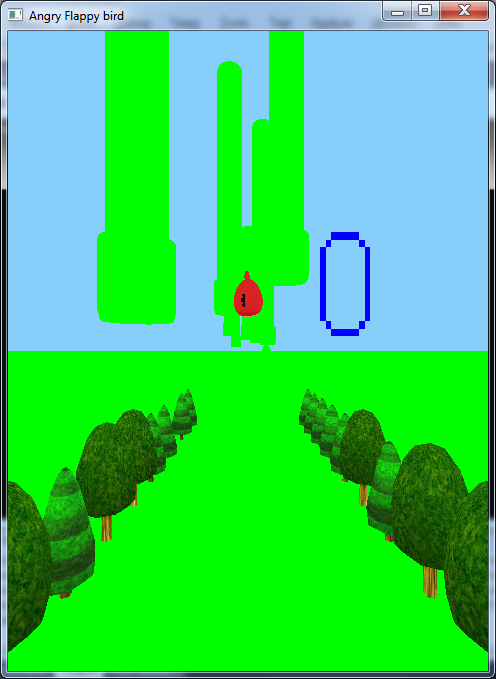
|  |
| --- |
| GLfloat ambient[] = { 0.2f, 0.2f, 0.2f };  GLfloat diffuse[] = { 0.8f, 0.8f, 0.8f };  GLfloat specular[] = { 0.0f, 0.0f, 0.0f };  GLfloat position[] = { -1.5f, 1.0f, -4.0f, 1.0f }; |

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：

