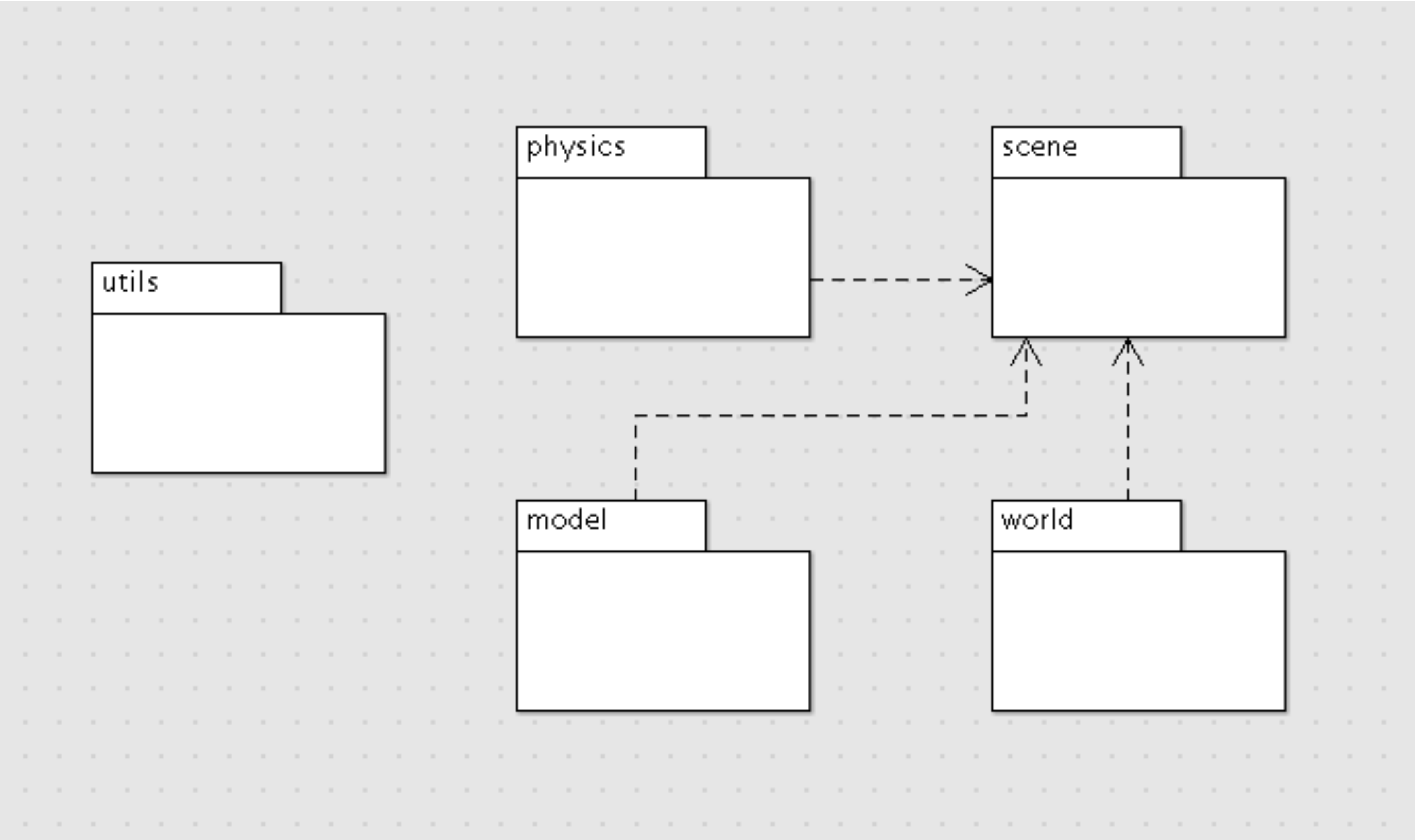
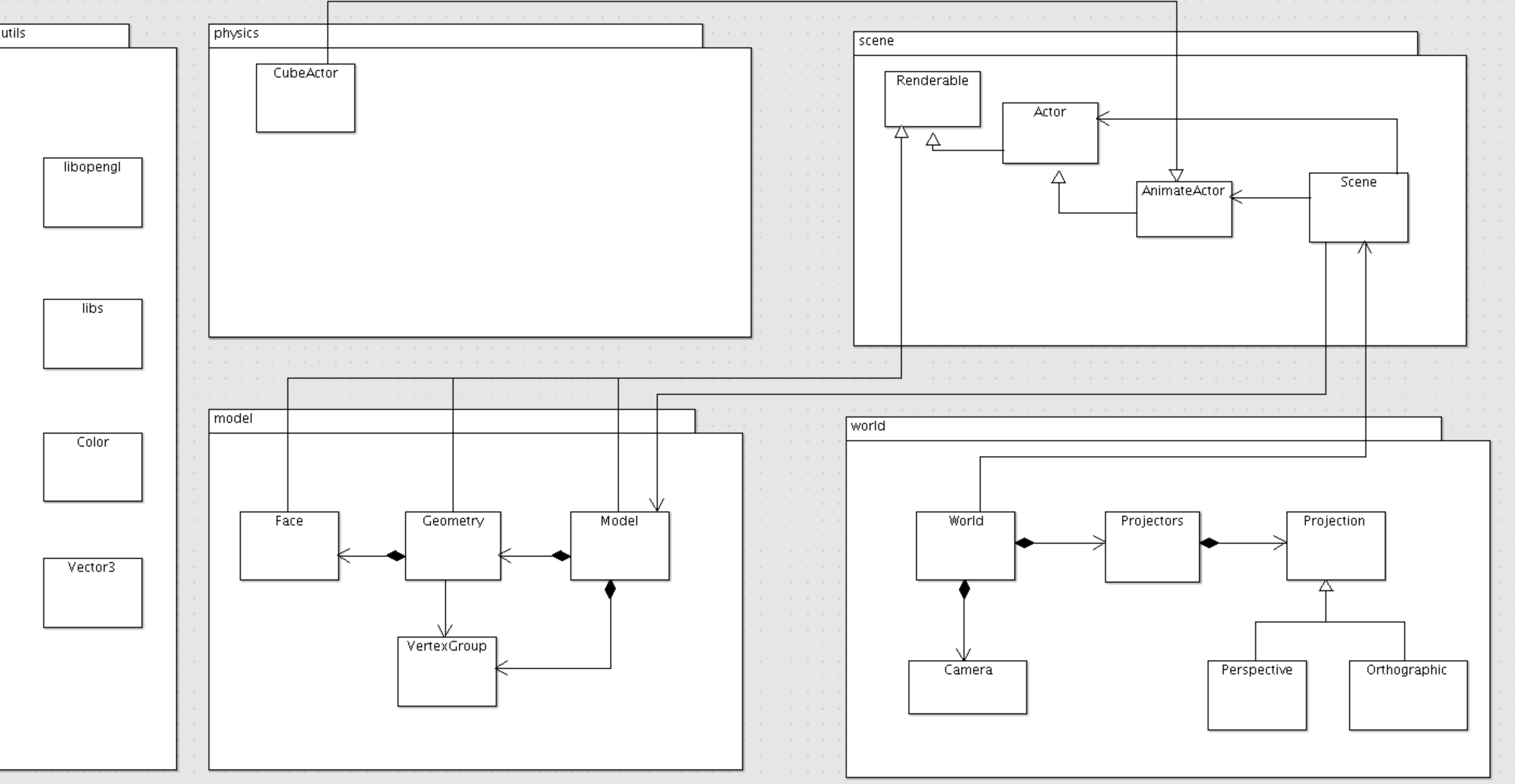
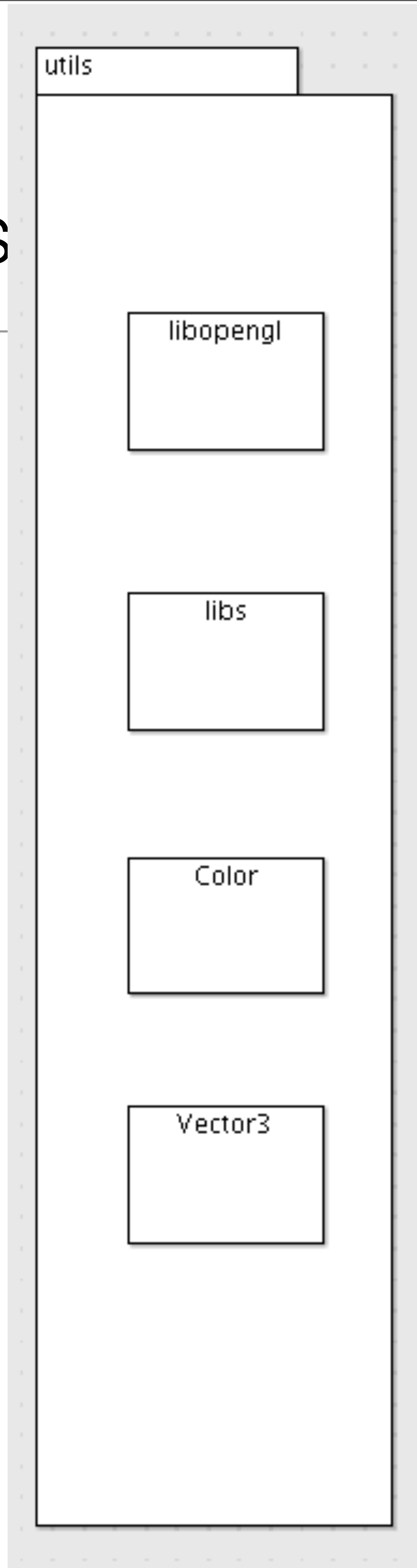
World Model

Architecture




utils
libopengl
libs
Color
Vector3
physics
CubeActor
scene
Renderable
Actor
AnimateActor
Scene
model
Face
Geometry
Model
VertexGroup
world
World
Projectors
Projection
Camera
Perspective
Orthographic

Utils



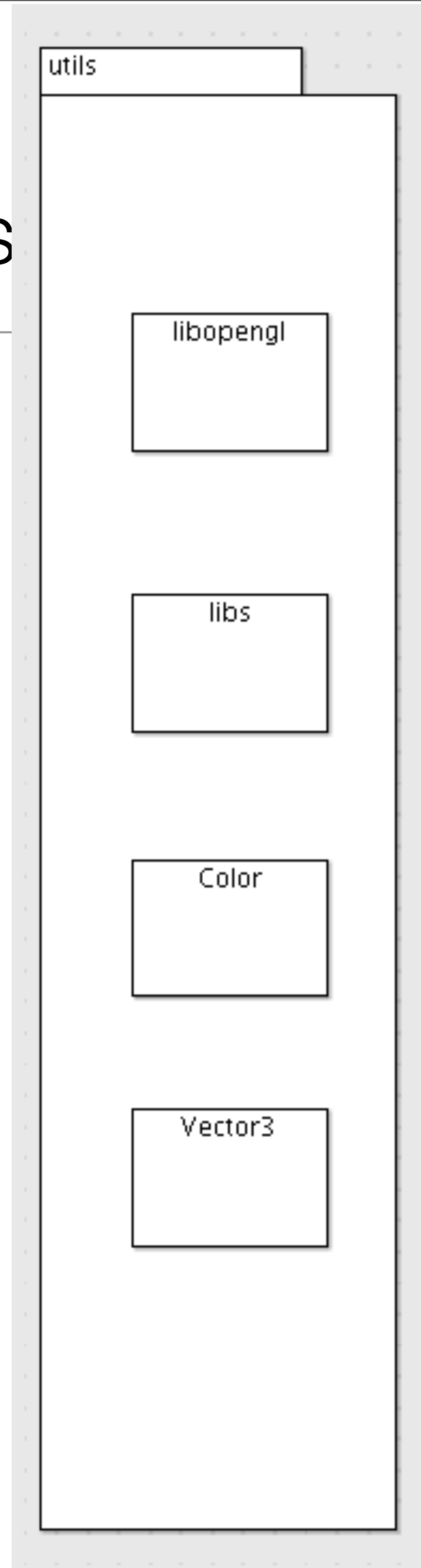
libopengl.h

```
#ifdef WIN32
#include <windows.h>
#include <gl\glu.h>
#include "freeglut.h"
#endif

#ifdef __APPLE__
#include "gl.h"
#include "glu.h"
#include "glut.h"
#endif
```

libs.h

```
#pragma once

#include "libopengl.h"

#include <boost/foreach.hpp>
#define foreach BOOST_FOREACH

#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

#include <vector>
#include <map>

#include <boost/ptr_container/ptr_map.hpp>
```

Utils




```
struct Vector3
{
  float X;
  float Y;
  float Z;

  static Vector3 UnitX;
  static Vector3 UnitY;
  static Vector3 UnitZ;

  Vector3(float x, float y, float z);
  Vector3(float value);
  Vector3();
  Vector3(std::istream& is);

  void translate();
  void rotate (float angle);

  void render();

  static const Vector3 & zero();

  inline Vector3& operator= (const Vector3& rhs)
   ...
  inline Vector3& operator+= (const Vector3& rhs)
   ...
  inline Vector3& operator-= (const Vector3& rhs)
   ...
  inline Vector3& operator*= (float rhs)
   ...
  inline Vector3 operator+ (const Vector3 rhs) const
   ...
  inline Vector3 operator* (float rhs) const
   ...
  friend std::ostream& operator <<(std::ostream& outputStream, const Vector3& v);
};

inline Vector3 operator* (float lhs, const Vector3& rhs)
...
```

Utils

utils

libopengl

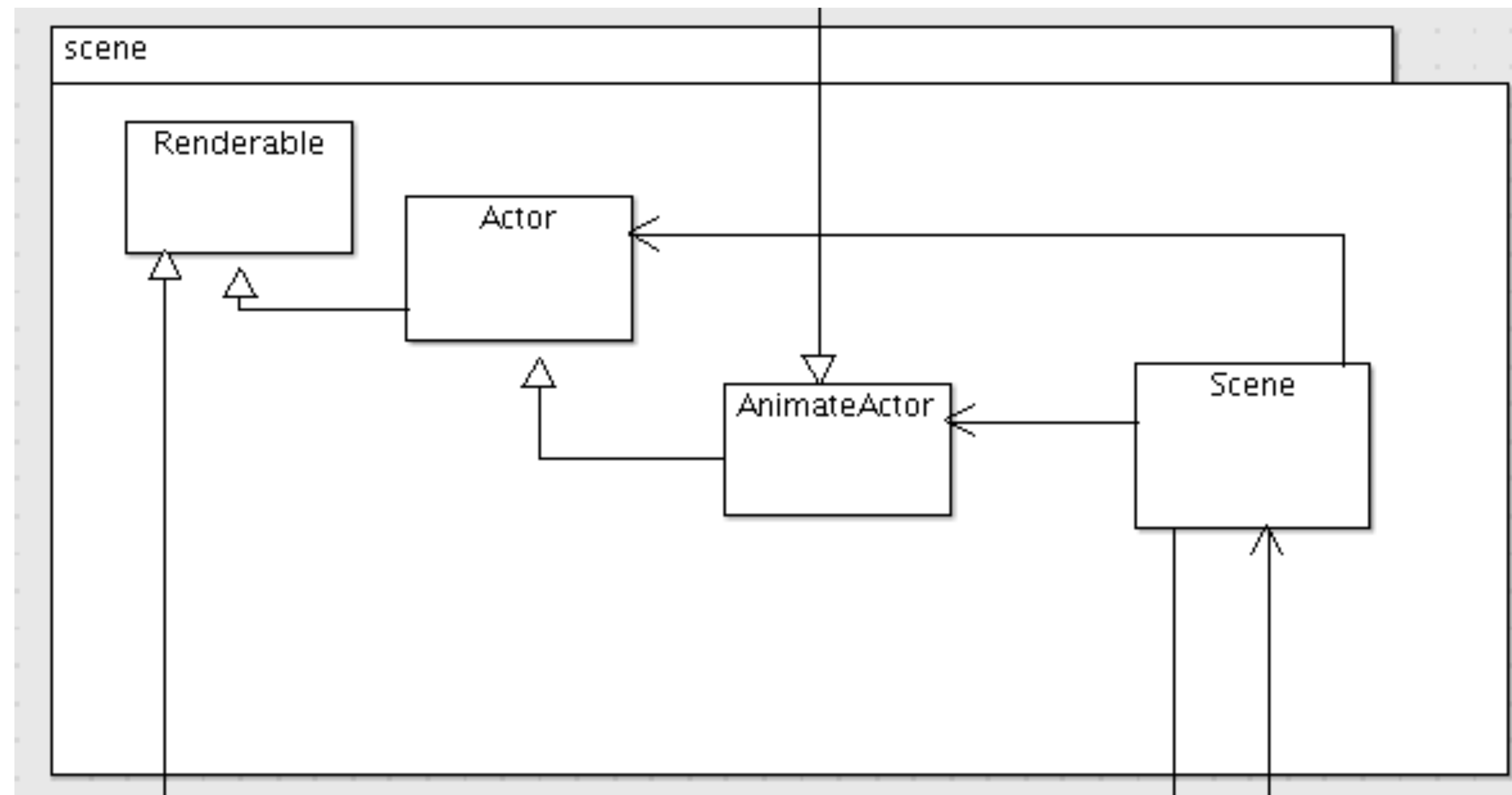
libs

Color

Vector3

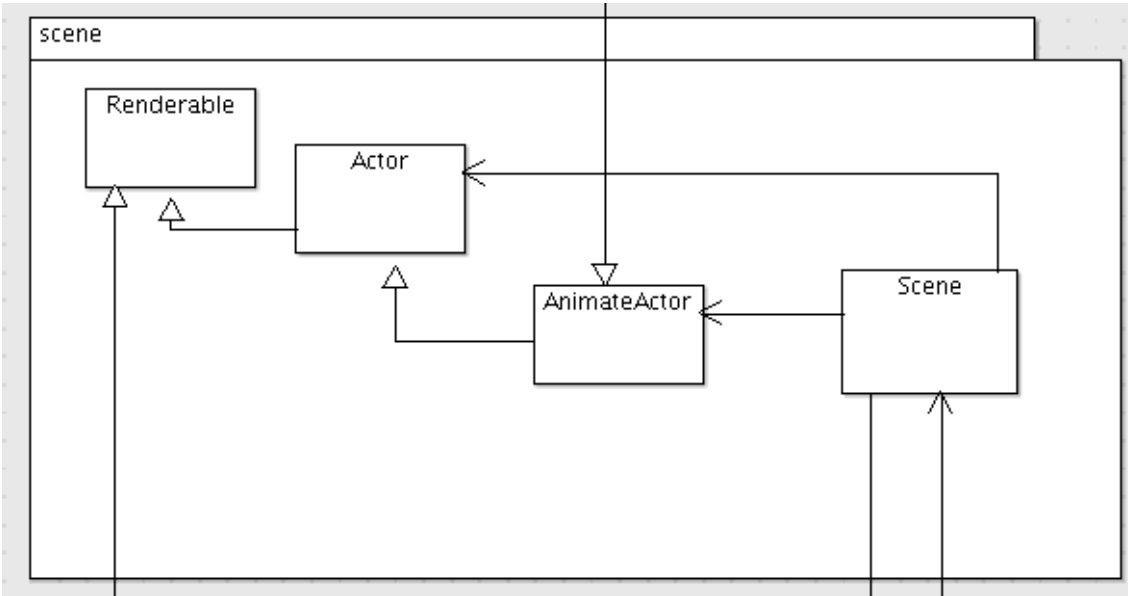
```
struct Color
{
  float R;
  float G;
  float B;
  float A;

  static Color White;
  static Color Yellow;
  static Color Red;
  static Color Magenta;
  static Color Cyan;
  static Color Green;
  static Color Black;
  static Color Blue;


  Color();
  Color(float r, float g, float b, float a=1.0f);
  Color(int r, int g, int b, int a=255);

  void render();
  void renderClear();
};
```

Scene



Actors

```
struct Renderable
{
  virtual void render()=0;
};

typedef boost::ptr_map<std::string, Renderable> RenderableMap;
```

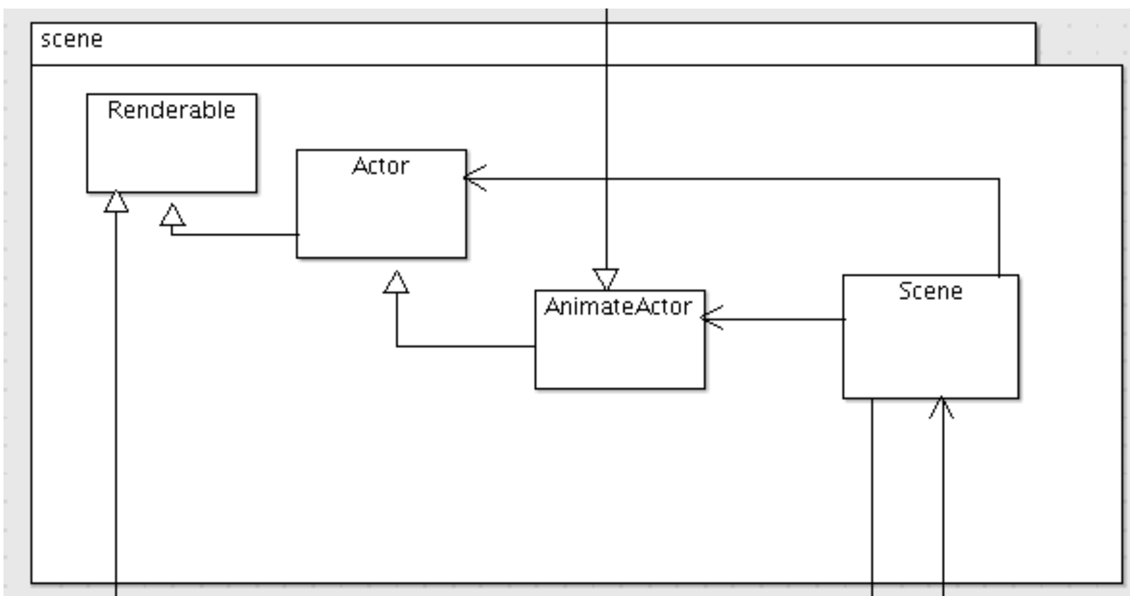



```
struct Actor : public Renderable
{
  Geometry *geometry;

  Actor(Geometry* g) : geometry(g)
  {}

  void render()
  { geometry->render(); }
};

typedef boost::ptr_map<std::string, Actor> ActorMap;
```

```
struct AnimateActor : public Actor
{
  AnimateActor(Geometry* geometry) : Actor (geometry)
  {}

  virtual void integrate(float dt)=0;
};

typedef std::map<std::string, AnimateActor*> AnimateActorMap;
```

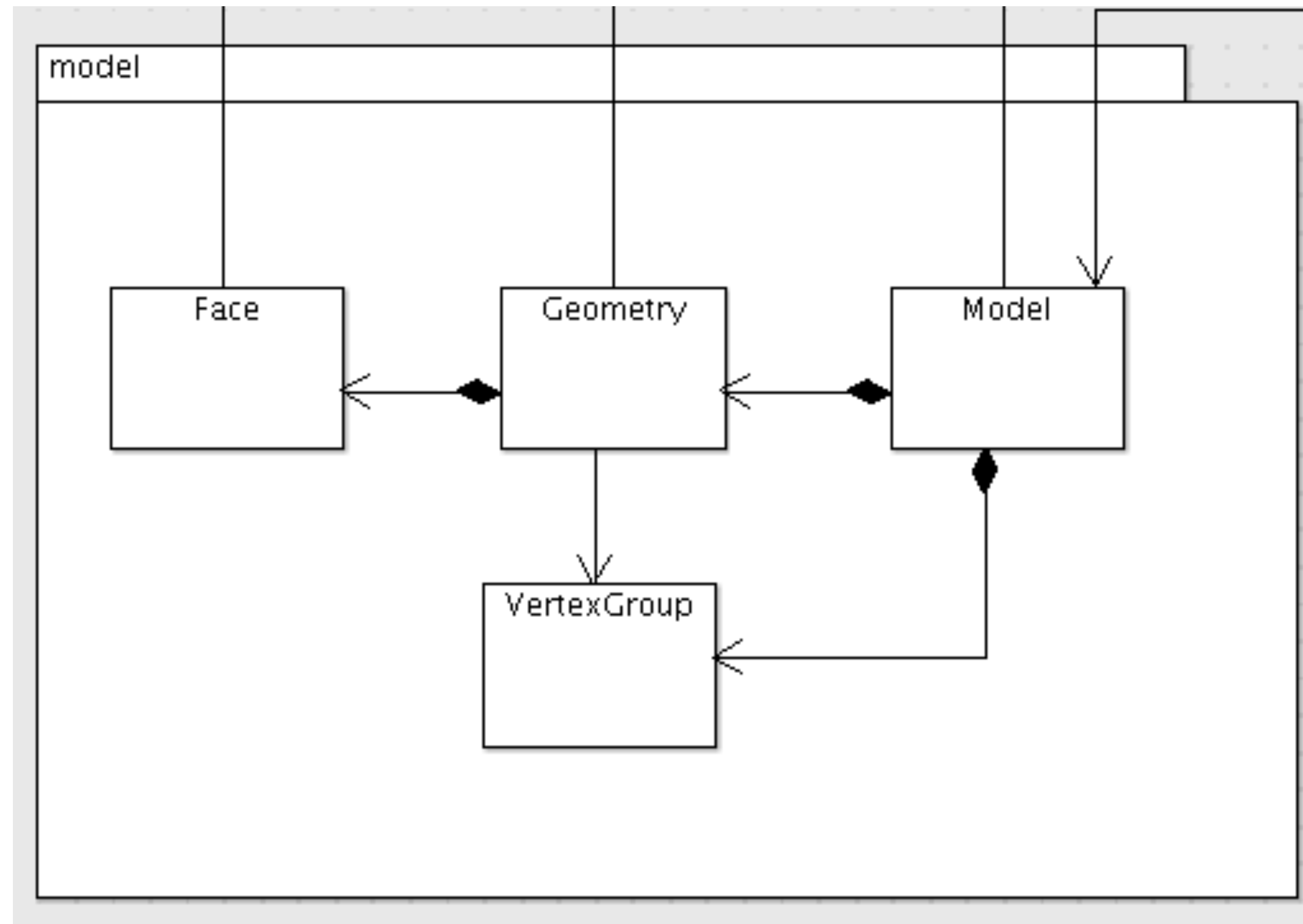
Scene

```
struct Scene
{
  Scene(Model*);

  void render();
  void tick(float secondsDelta);

  ActorMap          actors;
  AnimateActorMap   animateActors;
};
```



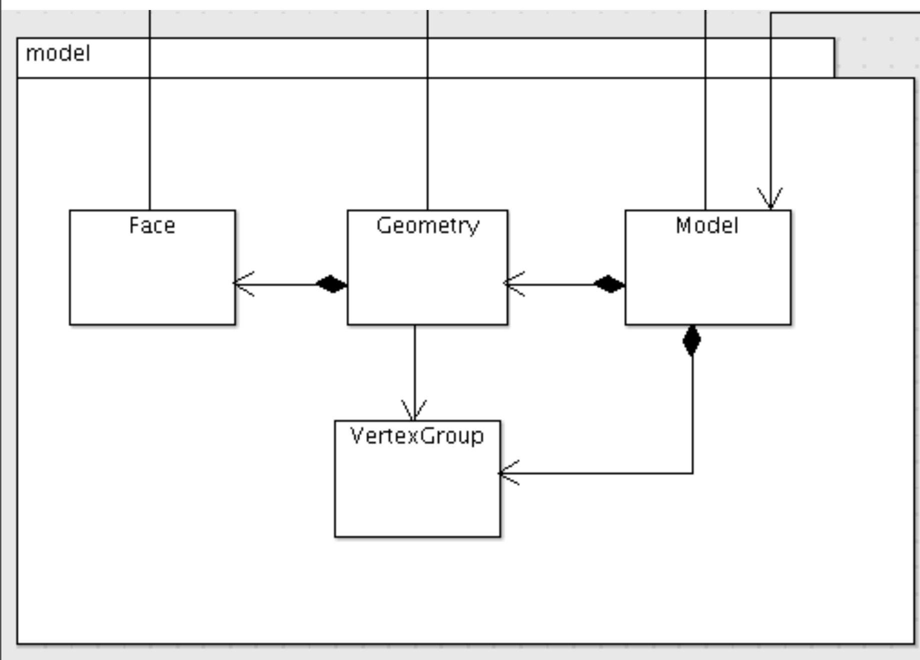

```
Scene:: Scene(Model *model)
{
  foreach (GeometryMap::value_type &value, model->entities)
  {
    string name = value.first;
    Actor *actor;
    if (name == "cube")
    {
      actor = new CubeActor(&value.second);
      animateActors[name] = (AnimateActor*) actor;
    }
    else
    {
      actor = new Actor(&value.second);
    }
    actors.insert(name, actor);
  }
}

void Scene::render()
{
  foreach (ActorMap::value_type value, actors)
  {
    value->second->render();
  }
}

void Scene::tick(float secondsDelta)
{
  foreach (AnimateActorMap::value_type value, animateActors)
  {
    value.second->integrate(secondsDelta);
  }
}
```

Model



Face

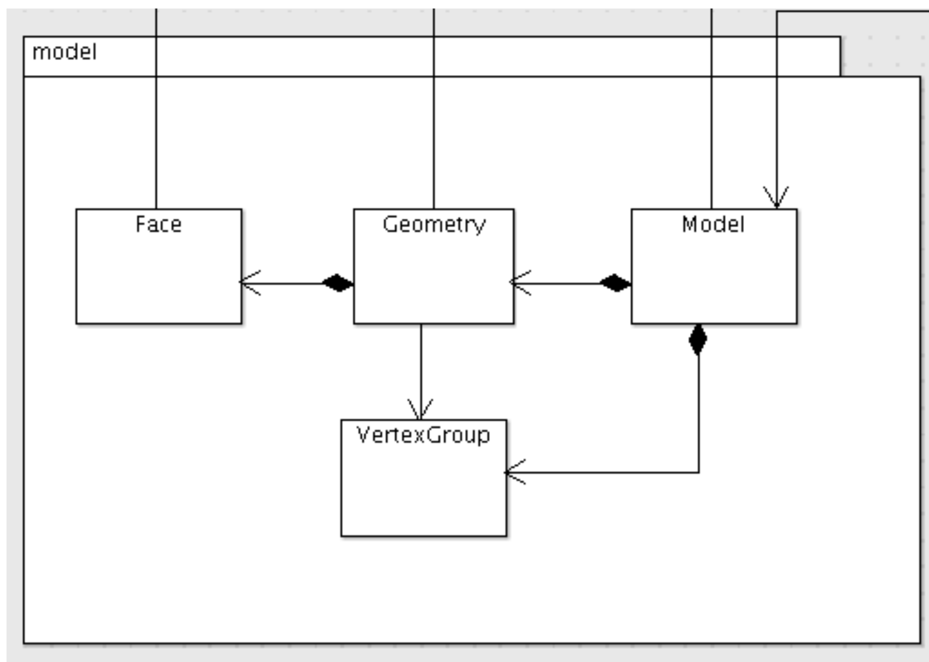
```
struct Face
{
  std::vector<int> vertexIndices;
  std::vector<int> textureIndices;

  Face(std::istream& is);
  void render(std::vector <Vector3>&);
};
```




```
Face::Face(istream& is)
{
  string line;
  getline (is, line);

  stringstream allIndexSets (line);
  string singleIndexSet;

  while( getline(allIndexSets, singleIndexSet, ' ') )
  {
   if (singleIndexSet.size() > 0)
   {
     string vertexIndex = singleIndexSet.substr(0, singleIndexSet.find('/'));
     int index = atoi(vertexIndex.c_str());
     vertexIndices.push_back(index);
   }
  }
}

void Face::render(std::vector <Vector3>&vertexTable)
{
  vertexIndices.size() == 3?
     glBegin(GL_TRIANGLES)
    :glBegin(GL_QUADS);

  foreach (int index, vertexIndices)
  {
    glVertex3f( vertexTable[index-1].X,
                vertexTable[index-1].Y,
                vertexTable[index-1].Z );
  }
  glEnd();
}
```

VertexGroup

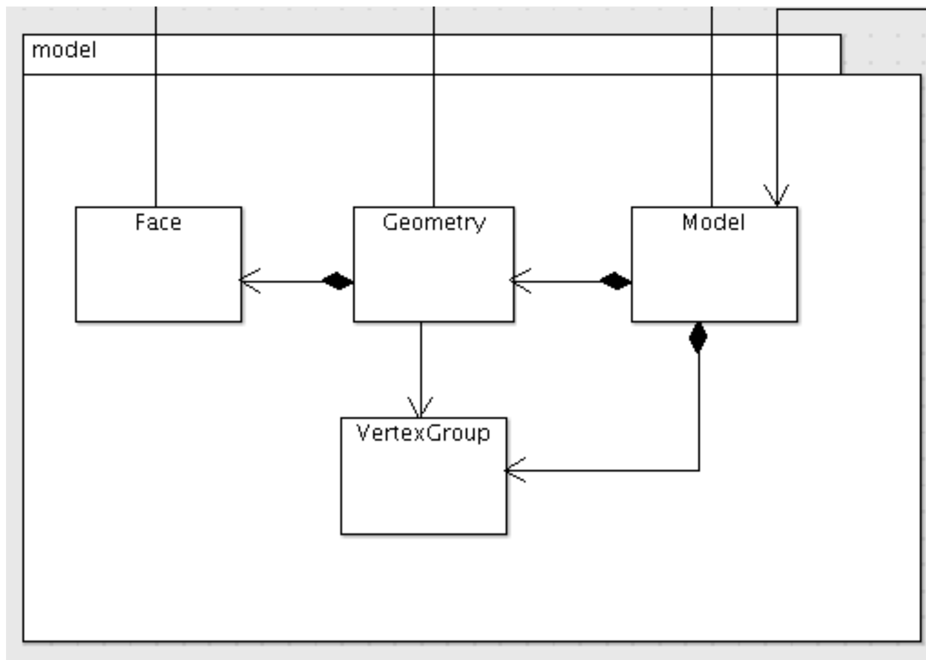
```
struct VertexGroup
{
  std::vector <Vector3> vertices;

  VertexGroup();
  void load(std::istream&);
};
```




```
VertexGroup::VertexGroup()
{}

void VertexGroup::load(istream& is)
{
  string indicator;
  bool stillGroup=true;
  do
  {
    is >> indicator;
    if (indicator == "v")
    {
      vertices.push_back(Vector3(is));
    }
    else if (indicator == "g")
    {
      stillGroup = false;
    }
    else
    {
      string buf;
      getline(is, buf);
    }
  } while (stillGroup && !is.eof());
  is.putback(indicator[0]);
}
```

Geometry

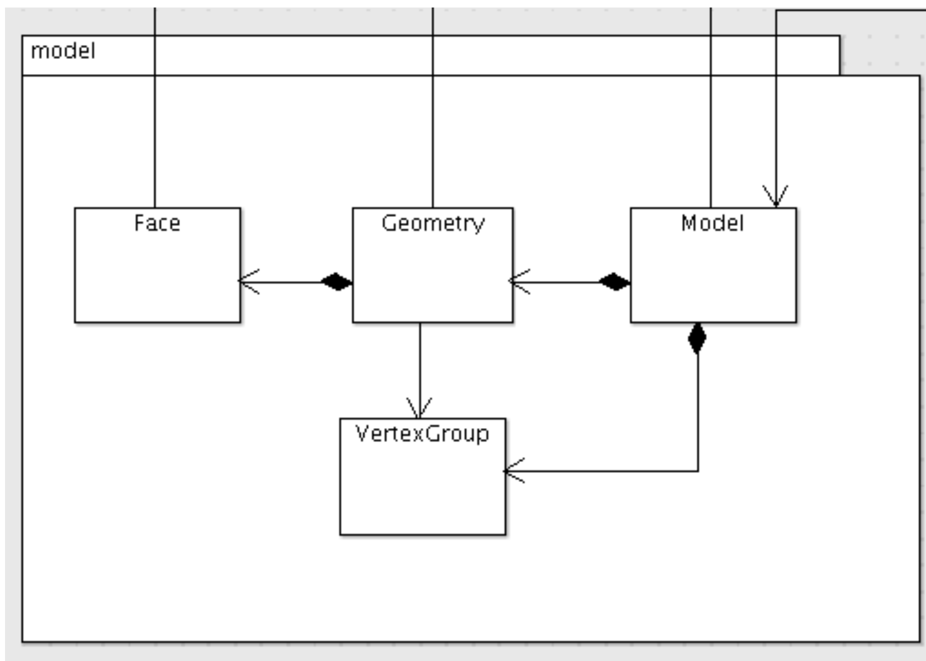
```
struct Geometry : public Renderable
{
  std::string name;
  std::vector<Face> faces;
  VertexGroup *vertexGroup;

  Geometry();
  Geometry(std::string name, std::istream&, VertexGroup*);
  void render();
};

typedef std::map <std::string, Geometry>  GeometryMap;
```




```
Geometry::Geometry()
{}

Geometry::Geometry(string groupName, istream& is,
VertexGroup*group)
: name(groupName), vertexGroup(group)
{
  string indicator;
  bool stillGroup=true;
  do
  {
    is >> indicator;
    if (indicator == "f")
    {
      faces.push_back(Face(is));
    }
    else if (indicator == "g")
    {
      stillGroup = false;
    }
    else
    {
      string buf;
      getline(is, buf);
    }
  } while (stillGroup && !is.eof());
  is.putback(indicator[0]);
}

void Geometry::render()
{
  foreach (Face &face, faces)
  {
    face.render(vertexGroup->vertices);
  }
}
```

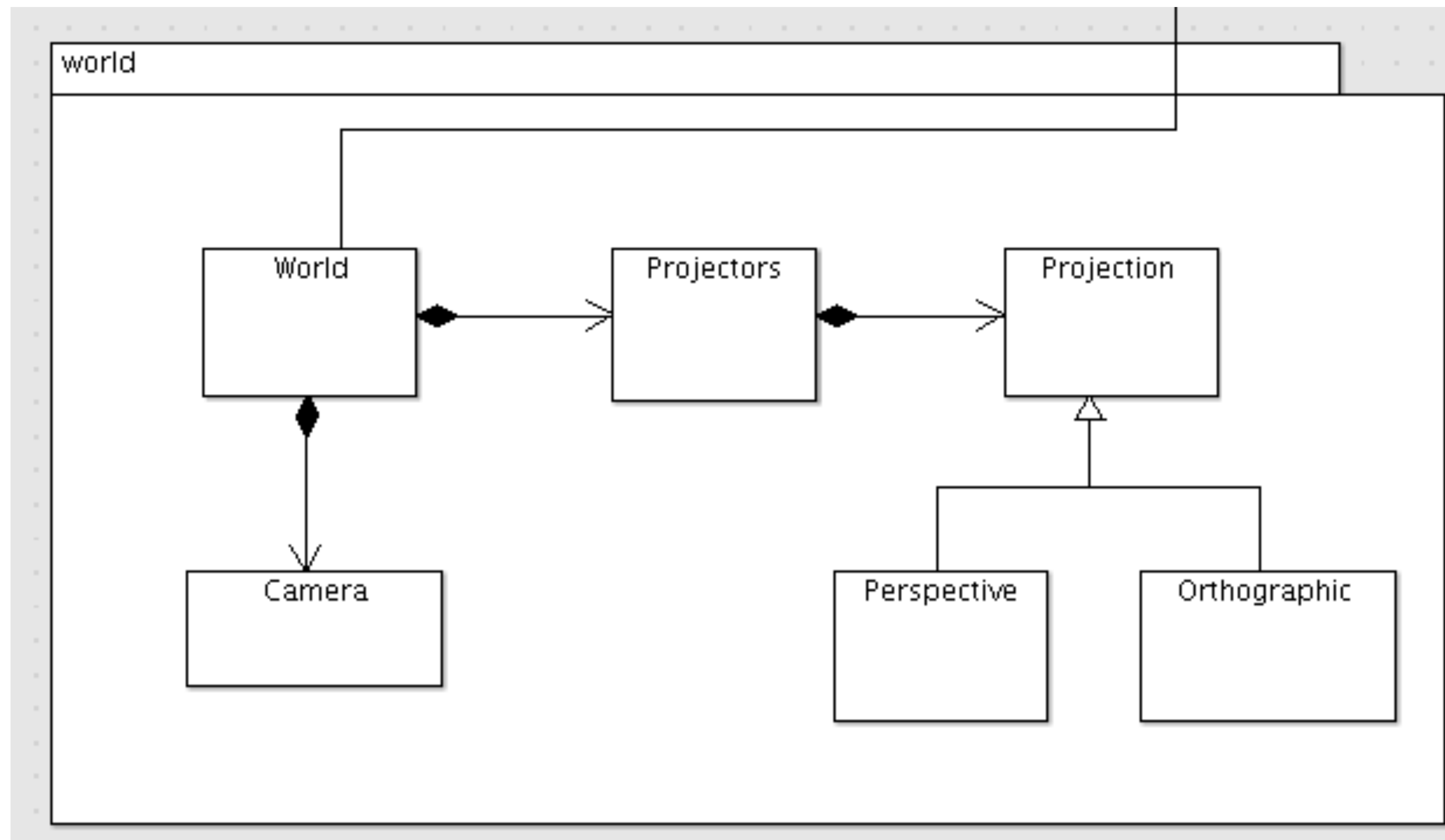
Model

```
struct Model : public Renderable
{
  GeometryMap entities;
  VertexGroup defaultGroup;

  Model();
  bool load(std::istream &is);
  void render();
};
```

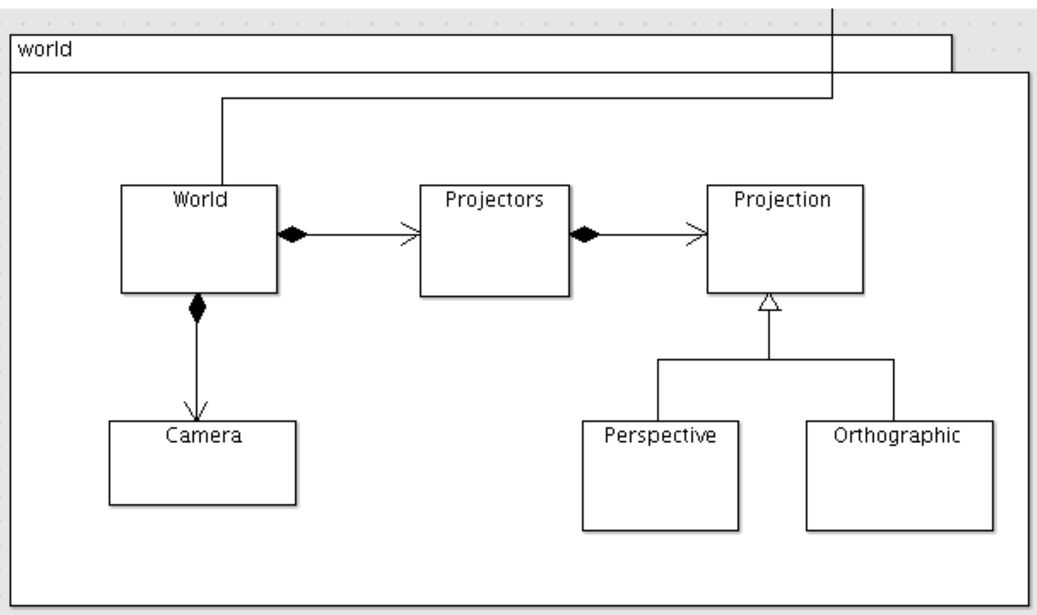



```
Model::Model()
{
}

bool Model::load(istream& is)
{
  string indicator;
  is >> indicator;
  while (!is.eof())
  {
    if (indicator == "#")
    {
      string buf;
      getline(is, buf);
    }
    else if (indicator == "g")
    {
      string name;
      is >> name;
      if (name == "default")
      {
        defaultGroup.load(is);
      }
      else
      {
        Geometry a(name, is, &defaultGroup);
        if (entities.find(a.name) == entities.end())
        {
          entities[a.name] = a;
        }
      }
    }
    is >> indicator;
  }
  return true;
}

void Model::render()
{
  foreach (GeometryMap::value_type &value, entities)
  {
    value.second.render();
  }
}
```

World



Projection

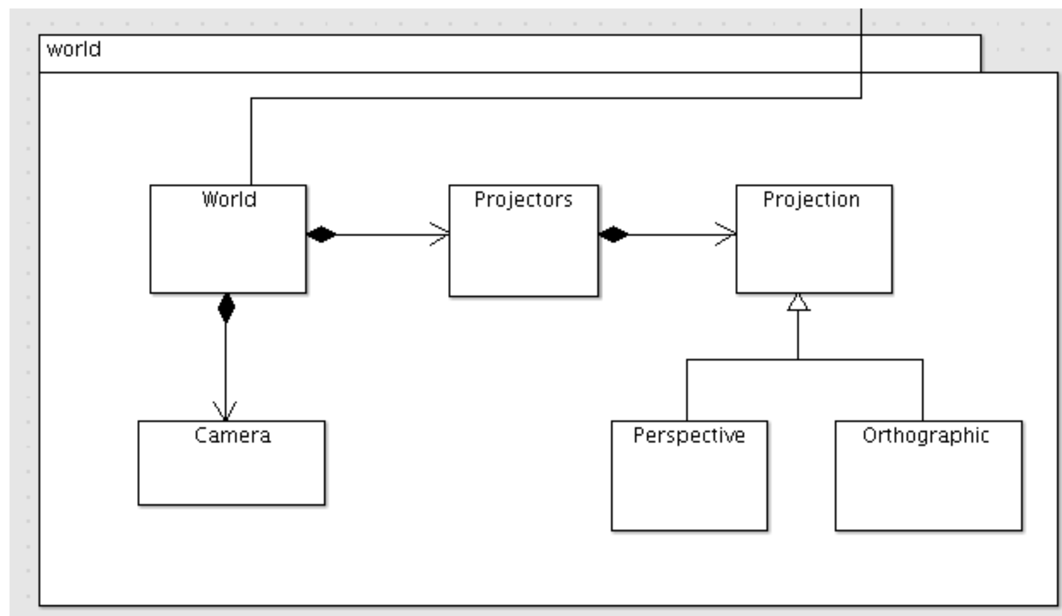



```
typedef std::pair<float, float> Range;

struct Projection
{
  Range windowSize;

  void resize(Range size);
  virtual void render()=0;
};

struct Orthographic: public Projection
{
  Range xRange;
  Range yRange;
  Range zRange;
  Vector3 axis;
  int angle;

  Orthographic(Range x, Range y, Range z, int angle, Vector3 axis);
  void render();
};

struct Perspective : public Projection
{
  float fovy;
  Range zRange;
  float zDistance;

  Perspective (float fovy, Range zRange, float zDistance);
  void render();
};

typedef boost::ptr_map <std::string, Projection>  ProjectionMap;
```

Projection Implementation

```
void Projection::resize(Range size)
{
  windowSize = size;
}

Orthographic::Orthographic(Range x, Range y, Range z, int theAngle, Vector3 theAxis)
: xRange(x), yRange(y), zRange(z), angle(theAngle), axis(theAxis)
{
}

void Orthographic::render()
{
  glLoadIdentity();
  glViewport(0, 0, windowSize.first, windowSize.second);
  glMatrixMode ( GL_PROJECTION);
  glLoadIdentity();
  glOrtho(xRange.first, xRange.second, yRange.first, yRange.second, zRange.first,
zRange.second);
  glMatrixMode ( GL_MODELVIEW);

  axis.rotate(angle);
}

Perspective::Perspective (float fovy, Range zRange, float zDistance)
: fovy(fovy), zRange(zRange), zDistance(zDistance)
{
}

void Perspective::render()
{
  glLoadIdentity();
  glViewport(0, 0, windowSize.first, windowSize.second);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(fovy, windowSize.first/windowSize.second, zRange.first, zRange.second);
  glMatrixMode (GL_MODELVIEW);
  Vector3(0,0,zDistance).translate();
}
```

Projectors

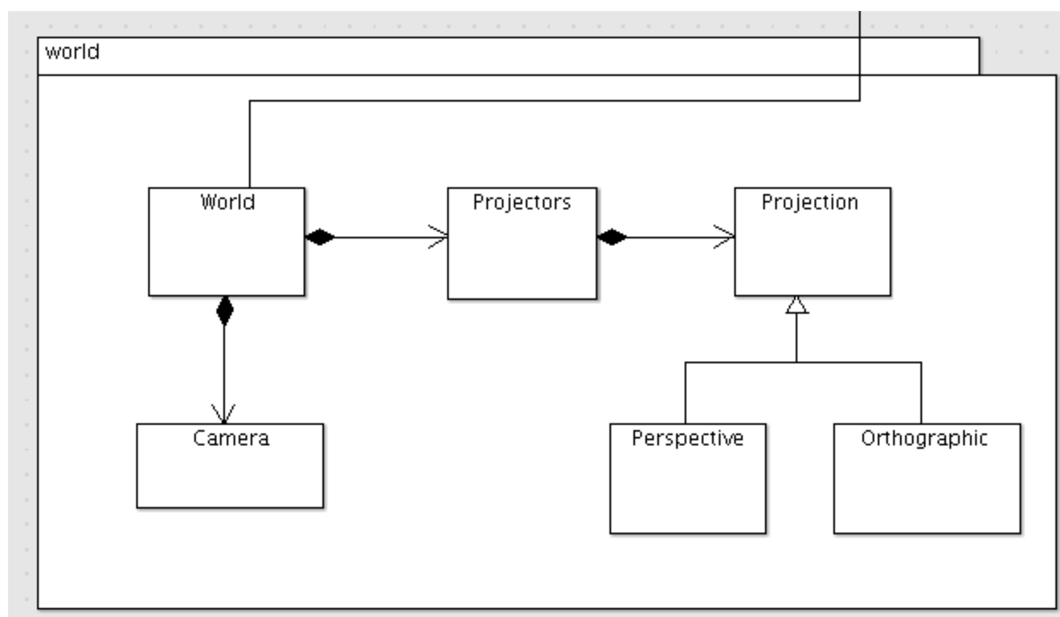



```
struct Projectors
{
  Projectors();

  bool isPerspective();
  void keyPress(unsigned char key);
  void windowReshape(int w, int h);

  void addProjection(std::string, Projection *projection);

  ProjectionMap projections;
  Projection   *currentProjection;
};
```

Projectors

```
Projectors::Projectors()
{
  currentProjection = new Perspective(60, Range(1,1000), -5);

  addProjection("1", currentProjection);
  addProjection("2", new Orthographic (Range(-10,10), Range(-10,10), Range(-10,10), 90, Vector3::UnitX));
  addProjection("3", new Orthographic (Range(-10,10), Range(-10,10), Range(-10,10), 90, Vector3::UnitY));
  addProjection("4", new Orthographic (Range(-10,10), Range(-10,10), Range(-10,10), 90, Vector3::UnitZ));
}

bool Projectors::isPerspective()
{
  Perspective *p = dynamic_cast<Perspective*> (currentProjection);
  if (p)
  {
    return true;
  }
  else
  {
    return false;
  }
}

void Projectors::keyPress(unsigned char ch)
{
  Range windowSize = currentProjection->windowSize;
  string projection;
  projection+=ch;
  ProjectionMap::iterator iter = projections.find(projection);
  if (iter != projections.end())
  {
    currentProjection = iter->second;
    currentProjection->resize(windowSize);
    currentProjection->render();
  }
}

void Projectors::addProjection(std::string str, Projection *projection)
{
  projections.insert(str, projection);
}

void Projectors::windowReshape(int w, int h)
{
  currentProjection->resize(Range(w,h));
  currentProjection->render();
}
```

Camera



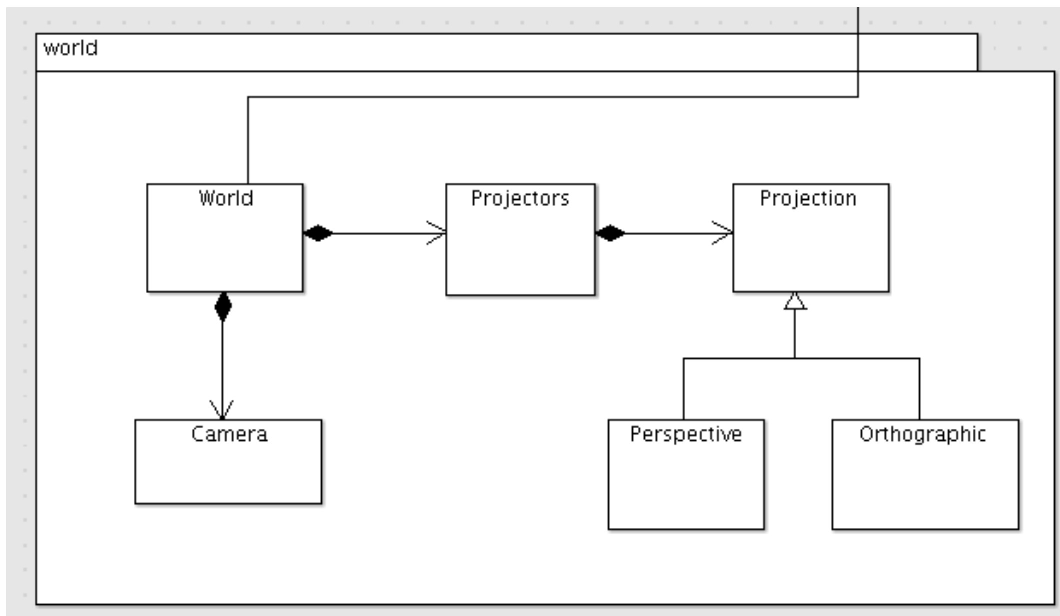

```
struct Camera
{
  Camera();
  void keyStroke (unsigned char key);
  void render();

  float xrot;
  float yrot;
  Vector3 position;
  float controlPrecision;
  float zoomPrecision;
};
```

Camera

```
Camera::Camera()
: position (0,0,-10)
{
  controlPrecision = 0.5;
  zoomPrecision = 0.1;
}

void Camera::render()
{
  Vector3::UnitX.rotate(xrot);
  Vector3::UnitY.rotate(yrot);
  position.translate();
}
```

```
void Camera::keyStroke (unsigned char key)
{
  if (key == 'q')
  {
    xrot += controlPrecision;
    if (xrot > 360)
      xrot -= 360;
  }
  else if (key == 'z')
  {
    xrot -= controlPrecision;
    if (xrot < -360)
      xrot += 360;
  }
  else if (key == 'd')
  {
    yrot += controlPrecision;
    if (yrot > 360)
      yrot -= 360;
  }
  else if (key == 'a')
  {
    yrot -= controlPrecision;
    if (yrot < -360)
      yrot += 360;
  }
  else if (key == 'w' || key == 's')
  {
    float xrotrad, yrotrad;
    yrotrad = (yrot / 180 * GL_PI);
    xrotrad = (xrot / 180 * GL_PI);
    if (key == 'w')
    {
      position.X += float(sin(yrotrad));
      position.Y -= float(sin(xrotrad));
      position.Z -= float(cos(yrotrad))  * zoomPrecision;
      //position.Z -= controlPrecision;
      cout << position.Z << endl;
    }
    else if (key == 's')
    {
      position.X -= float(sin(yrotrad));
      position.Y += float(sin(xrotrad));
      position.Z += float(cos(yrotrad)) * zoomPrecision;
      //position.Z += controlPrecision;
      cout << position.Z << endl;
    }
  }
```

World




```
#define theWorld World::GetInstance()

struct World
{
  void initialize(std::string name, int width, int height);
  void keyPress(unsigned char ch);
  void start();
  void render();
  void tickAndRender();

  static World& GetInstance();
  static World  *s_World;

  Scene         *scene;
  Projectors     projectors;
  Camera         camera;
};
```

World (1)

```
World& World::GetInstance()
{
  if (s_World == NULL)
  {
    s_World = new World();
  }
  return *s_World;
}

void World::initialize(string name, int width, int height)
{
  int argc=0;
  char** argv;
  glutInit(&argc, argv);

  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(width, height);
  glutCreateWindow(name.c_str());

  Color::Black.renderClear();
  glEnable(GL_DEPTH_TEST);
  glFrontFace(GL_CCW);
  glPolygonMode(GL_FRONT,GL_LINE);
  glPolygonMode(GL_BACK,GL_LINE);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0f, 1, 1.0, 1000.0);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glutKeyboardFunc(keyboard);
  glutReshapeFunc(reshape);
  glutDisplayFunc(renderScene);
}

void World::start()
{
  timerFunc(0);
  glutMainLoop();
}
```

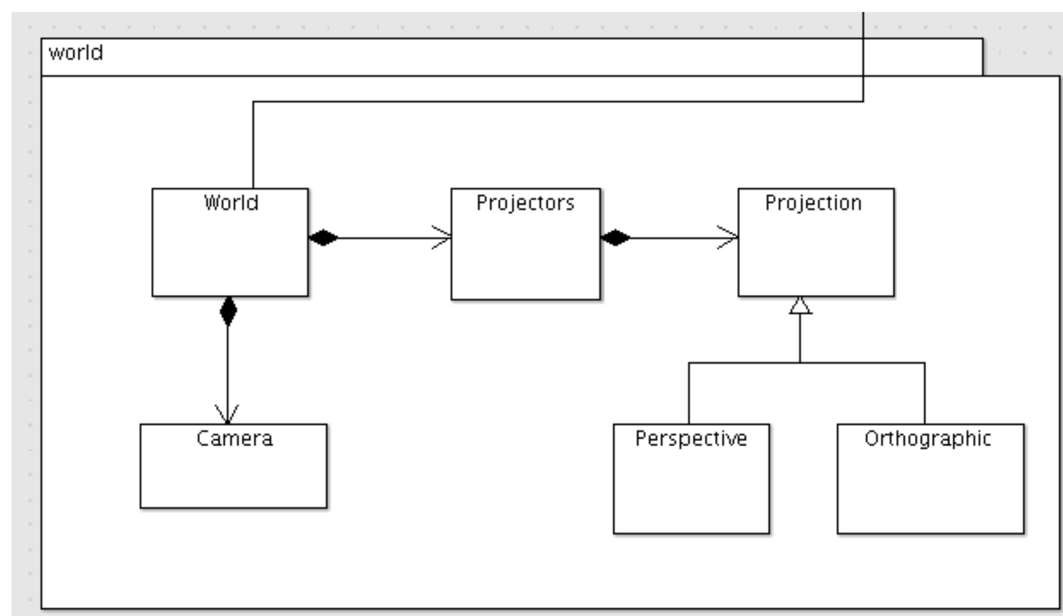
Glutadapter

```
void reshape(int w, int h);
void renderScene(void);
void keyboard(unsigned char key, int x, int y);
void timerFunc(int value);
```

```
void reshape(int w, int h)
{
  theWorld.projectors.windowReshape(w,h);
}

void renderScene(void)
{
  theWorld.render();
}

void keyboard(unsigned char key, int x, int y)
{
  theWorld.keyPress(key);
}

void timerFunc(int value)
{
  theWorld.tickAndRender();
  glutTimerFunc(50, timerFunc, 1);
}
```

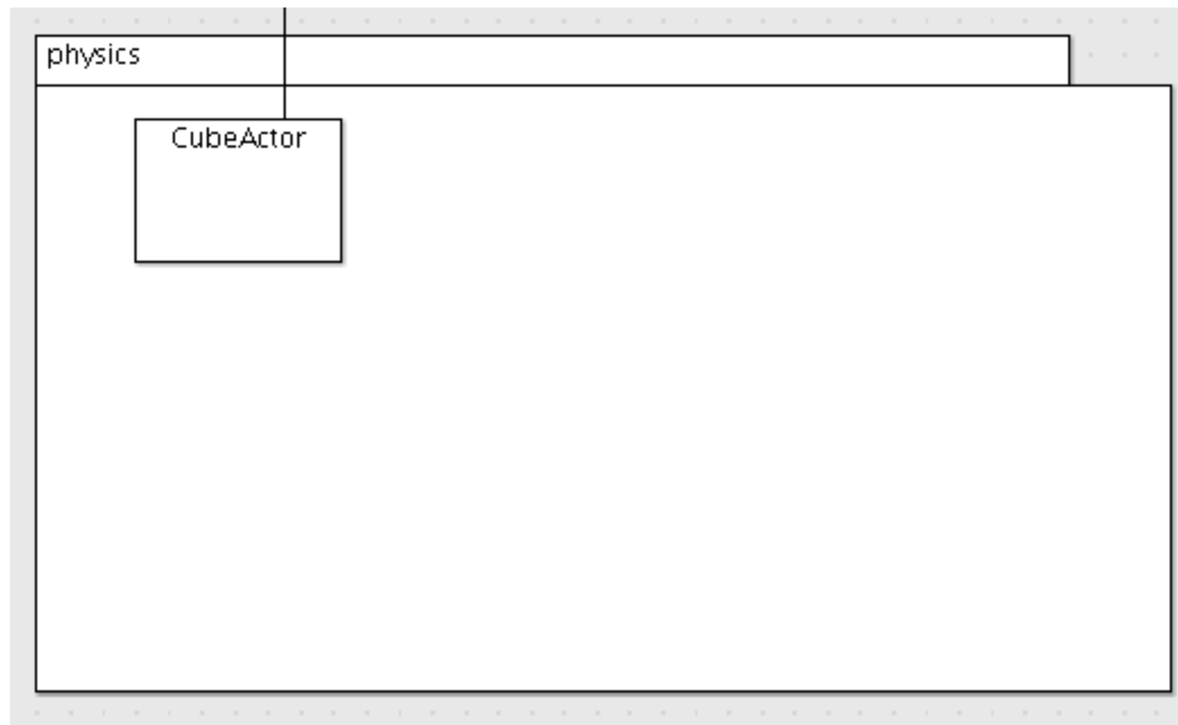
World (2)

```
void World::keyPress(unsigned char ch)
{
  if (ch >= '1' && ch <= '4')
  {
    projectors.keyPress(ch);
  }
  else
  {
    if (projectors.isPerspective())
    {
      camera.keyStroke(ch);
    }
  }
  glutPostRedisplay();
}
```




```
void World::render()
{
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  if (projectors.isPerspective())
  {
    glLoadIdentity();
    camera.render();
  }

  scene->render();

  glutSwapBuffers();
}

void World::tickAndRender()
{
  static clock_t lastTime = 0;

  if (lastTime == 0)
    lastTime = clock();

  clock_t currTime = clock();
  clock_t deltaTime = currTime - lastTime;
  float secondsDelta = (float)deltaTime/CLOCKS_PER_SEC;

  scene->tick(secondsDelta);

  glutPostRedisplay();
}
```

Physics

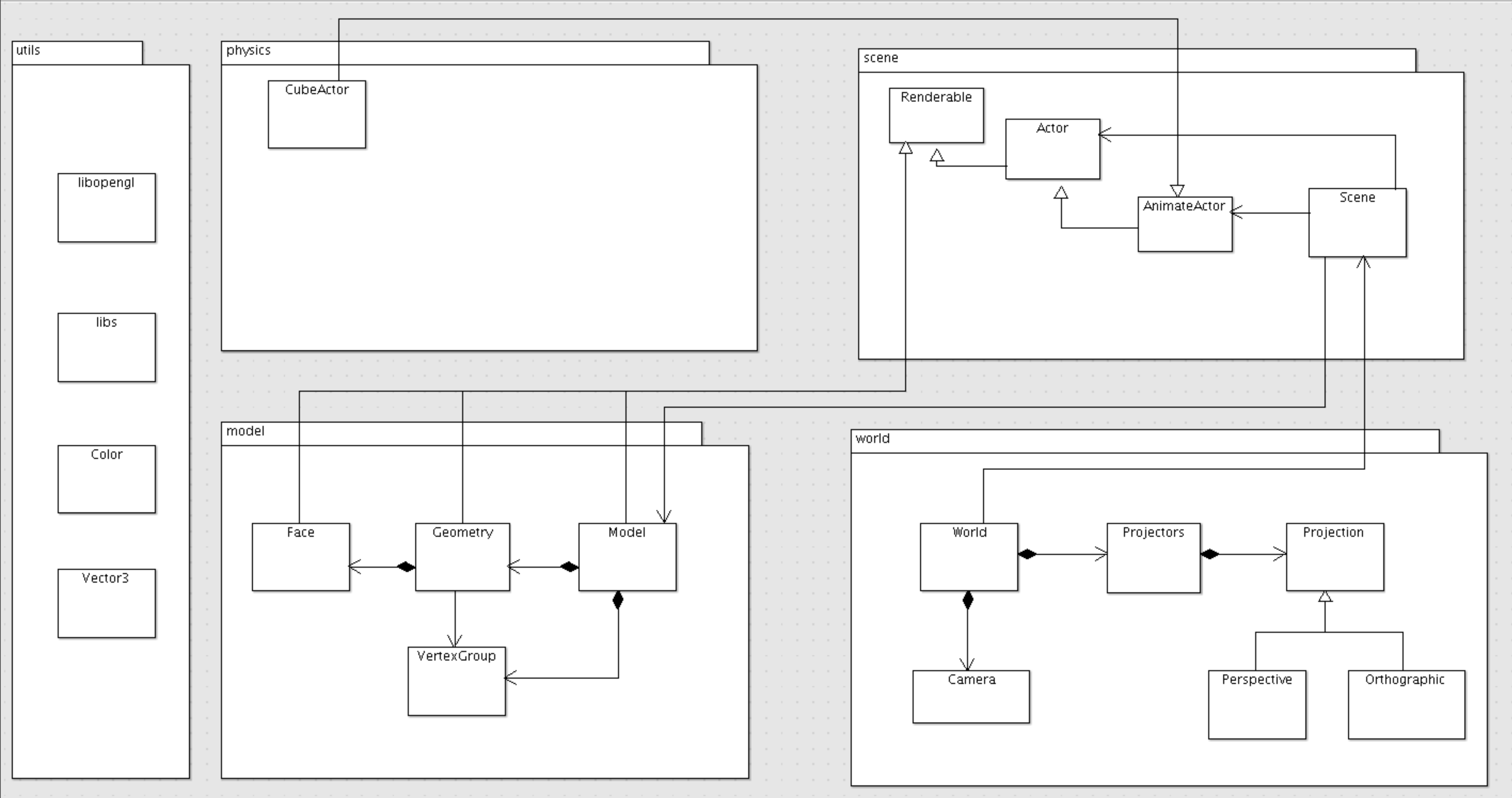



```
struct CubeActor : public AnimateActor
{
  Vector3 position;
  Vector3 velocity;
  Vector3 acceleration;

  Vector3 force;

  float inverseMass;
  float damping;

  CubeActor(Geometry *);

  void integrate(float dt);
  void render();
};
```

Physics

```
CubeActor::CubeActor(Geometry *geometry)
: AnimateActor(geometry)
{
  position = Vector3::zero();
  foreach (Face &face, geometry->faces)
  {
    foreach (int index, face.vertexIndices)
    {
      Vector3 vector = geometry->vertexGroup->vertices[index-1];
      position += vector;
    }
  }
  position *= 1.0f/8.0f;
  velocity = Vector3(0,1,0);
  acceleration = Vector3(0,-.1,0);
  inverseMass = 1.0f;
  damping = 1.0f;

  foreach (Face &face, geometry->faces)
  {
    foreach (int index, face.vertexIndices)
    {
      geometry->vertexGroup->vertices[index-1] -= position;
    }
  }
}
```

```
void CubeActor::integrate(float dt)
{
  // An unmovable particle has zero inverseMass.
  if (inverseMass <= 0.0f) return;

  // Work out the acceleration from the force.
  Vector3 resultingAcceleration(acceleration +
force*inverseMass);

  // Update linear velocity from the acceleration.
  velocity +=  dt*resultingAcceleration;

  // Impose artificial drag.
  velocity *= pow(damping, dt);

  // Update linear position.
  position += dt*velocity;

  // Clear the forces.
  force = Vector3::zero();

  return;
};

void CubeActor::render()
{
  glPushMatrix();
    glTranslatef(position.X, position.Y, position.Z);
    Actor::render();
  glPopMatrix();
}
```


utils
libopengl
libs
Color
Vector3
physics
CubeActor
scene
Renderable
Actor
AnimateActor
Scene
model
Face
Geometry
Model
VertexGroup
world
World
Projectors
Projection
Camera
Perspective
Orthographic

Grading Spectrum (Indicative only, to be updated...)

Range	*UML Model: 10%*	*Model: 30%*	*Rendering 30%*	*Physics: 30%*
Baseline	Simple class diagrams	Multiple Entities	Perspective + Multiple Orthographic Views	Determine Properties of some Physical Objects (Cube, sphere, position, orientation)
Good	+Packages	Textures & Materials	Windowed Display (shows Perspective + 3 Orthographic like Maya)	Simulations involving forces (gravity, spring)
Excellent	+Sequence	Lighting	Navigable Camera	Simulations involving forces with constrains (e.g. Newtons cradle)
Outstanding	+Activity diagrams	Aggregate Multiple Model Files	Scriptable Camera (proceed on some trajectory)	Collisions