Revised Scene Classes

Objectives

- Factor out physics related features from Scene into a new AnimateScene class

Scene (current)

- Incorporates basic scene rendering
  - + Animation support
  - + Force Registry

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

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

  ActorMap          actors;
  AnimateActorMap   animateActors;

  ForceGeneratorRegistry forceGeneratorRegistry;
};
```

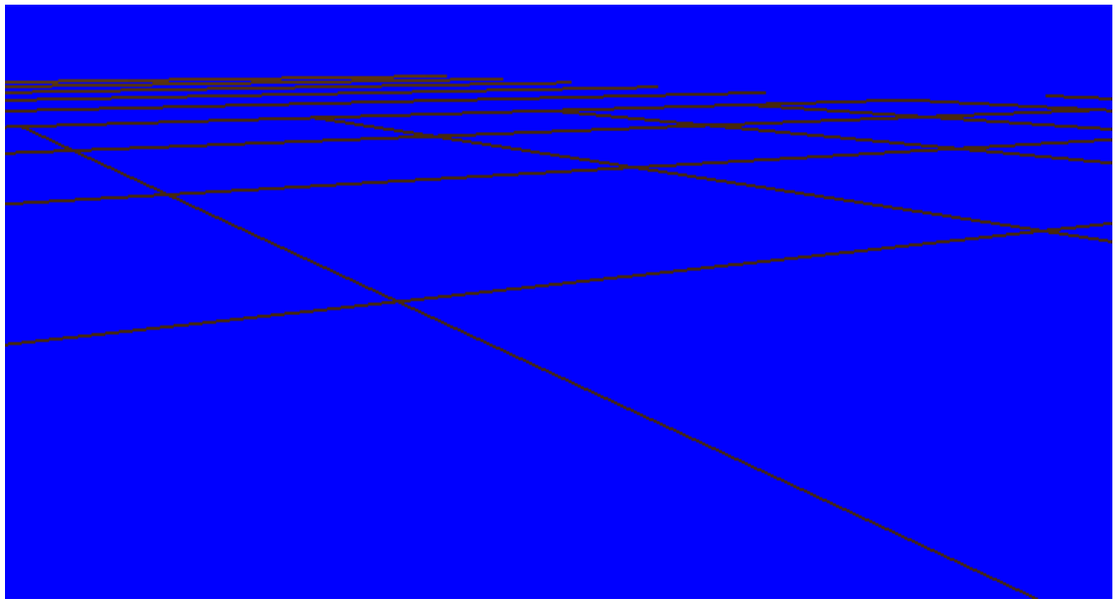
Scene (implementation)

- Clearly overburdened

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

  // draw grid on ground
  const float RANGE = 15.0f;
  glLineWidth(5);
  glColor3f(0.0, 1.0, 0.0);
  glBegin(GL_LINES);
  for (int i = -5; i <= 5; ++i)
  {
  float offset = 0.2 * float(i) * RANGE;
  glVertex3f(-RANGE, 0, offset);
  glVertex3f(RANGE, 0, offset);
  glVertex3f(offset, 0, -RANGE);
  glVertex3f(offset, 0, RANGE);
  }
  glEnd();
  glLineWidth(1);
}

void
Scene::tick(float secondsDelta)
{
  forceGeneratorRegistry.applyForce(secondsDelta);

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

```
Scene::Scene(Model *model)
{
  foreach (GeometryMap::value_type &value, model->entities)
  {
    string name = value.first;
    Actor *actor;
    std::cout << name << std::endl;

    // TODO (for you)
    // Determine physical primatives using name format pShape#
    //
    // String comparision should not include object number!!

    if (name == "pCube1")
    {
      actor = new CubeActor(&value.second);

    }
    else if (name == "pSphere1")
    {

      PhysicsActor * sphere = new SphereActor(&value.second);
      actor = dynamic_cast<Actor*>(sphere);

      // create an anchor
      Vector3 anchor = sphere->position + Vector3(0, 5, 0);
      // create force generator
      AnchoredSpringForceGenerator * fg = new AnchoredSpringForceGenerator(
        anchor, 1.0, 3.0);
      // add actor<->forceGenerator pair to registry
      forceGeneratorRegistry.add(sphere, fg);

    }
    else if (name == "pSphere2" || name == "pSphere3")
    {

      actor = new SphereActor(&value.second);

    }
    else
    {
      actor = new Actor(&value.second);
    }

    // add phhysics actors (name start wiath a 'p') to animateActors collection
    if (name[0] == 'p')
    {
      animateActors[name] = (AnimateActor*) actor;
    }

    actors.insert(name, actor);
  }
}
```

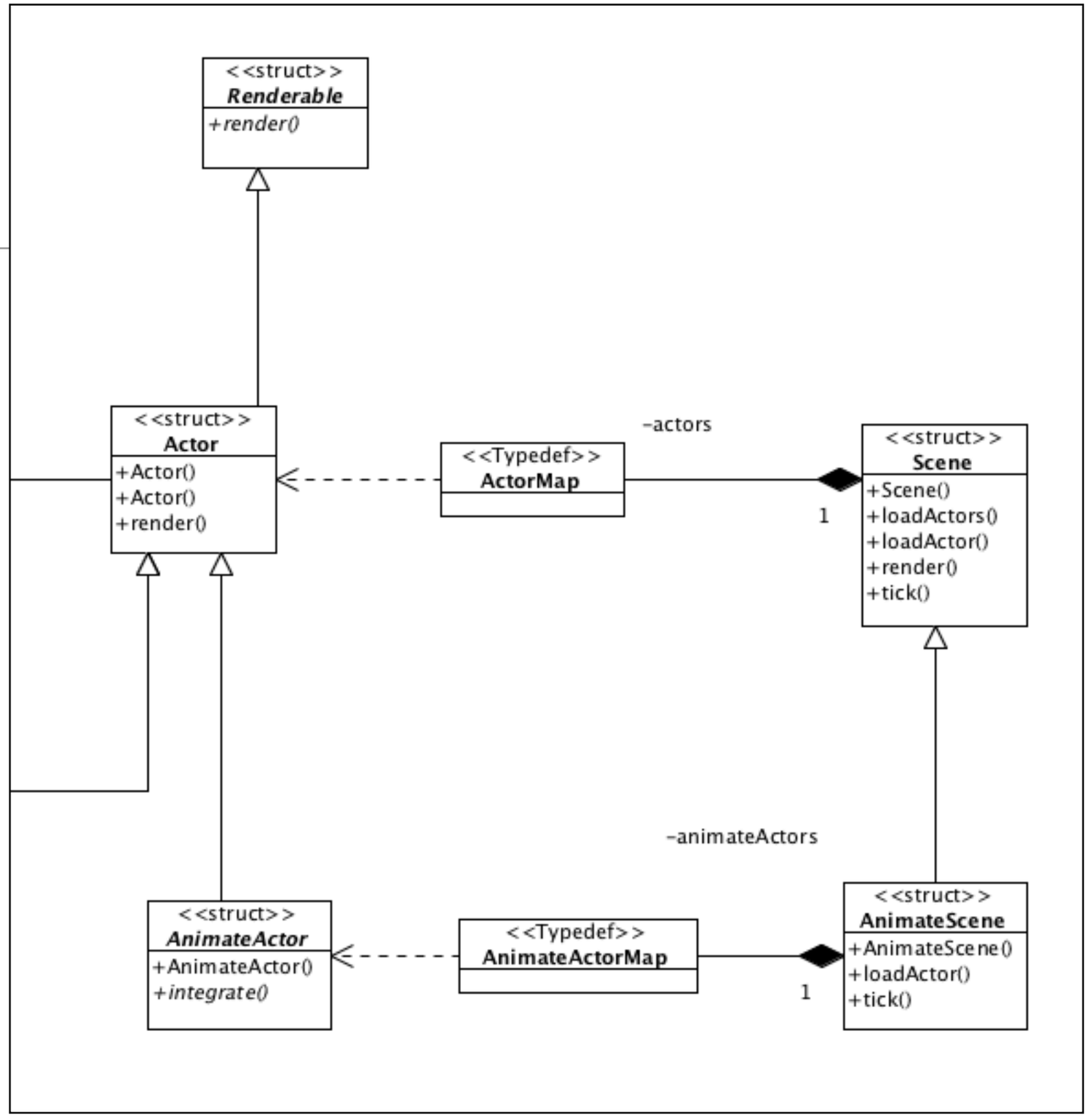
Factor out Grid class



```
struct Grid: public Actor
{
  Grid()
  {}

  void render()
  {
    const float RANGE = 15.0f;
    glLineWidth(2);
    glColor3f(0.0,1.0,0.0);
    glBegin(GL_LINES);
    for(int i=-10; i<=10; ++i)
    {
      float offset = 0.2*float(i)*RANGE;
      glVertex3f(-RANGE, 0, offset);
      glVertex3f( RANGE, 0, offset);
      glVertex3f( offset, 0,-RANGE);
      glVertex3f( offset, 0, RANGE);
    }
    glEnd();
    glLineWidth(1);
  }
};
```

Revised Model

- Scene base class has no knowledge of physics or AnimateActors
- Single task is to load and render a model.
- Can be used standalone
- AnimateScene incorporates physics mechanisms



Scene

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

  virtual Actor* loadActor(GeometryMap::value_type &value);
  virtual void render();
  virtual void tick(float secondsDelta) {}

  ActorMap          actors;
};
```

- loadActors() will load all actors from a given model

- loadActor() - is a ‘template method’ - and can be overridden by derived Scene implementation

Scene

- loadActors calls ‘loadActor’
- Default implementation loads a simple (non-physics) actor
- This can be overridden by derived class

```
Scene:: Scene()
{
}

void Scene::loadActors(Model*model)
{
  string actorName;
  foreach (GeometryMap::value_type &value, model->entities)
  {
    Actor* actor = loadActor(value);
    if (actor)
    {
      actorName = value.first;
      actors.insert(actorName, actor);
    }
  }
  actorName = "grid";
  actors.insert (actorName, new Grid());
  actorName = "jet";
  actors.insert (actorName, new JetPlane());
}

Actor* Scene::loadActor(GeometryMap::value_type &value)
{
  return new Actor(&value.second);
}

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

AnumateScene

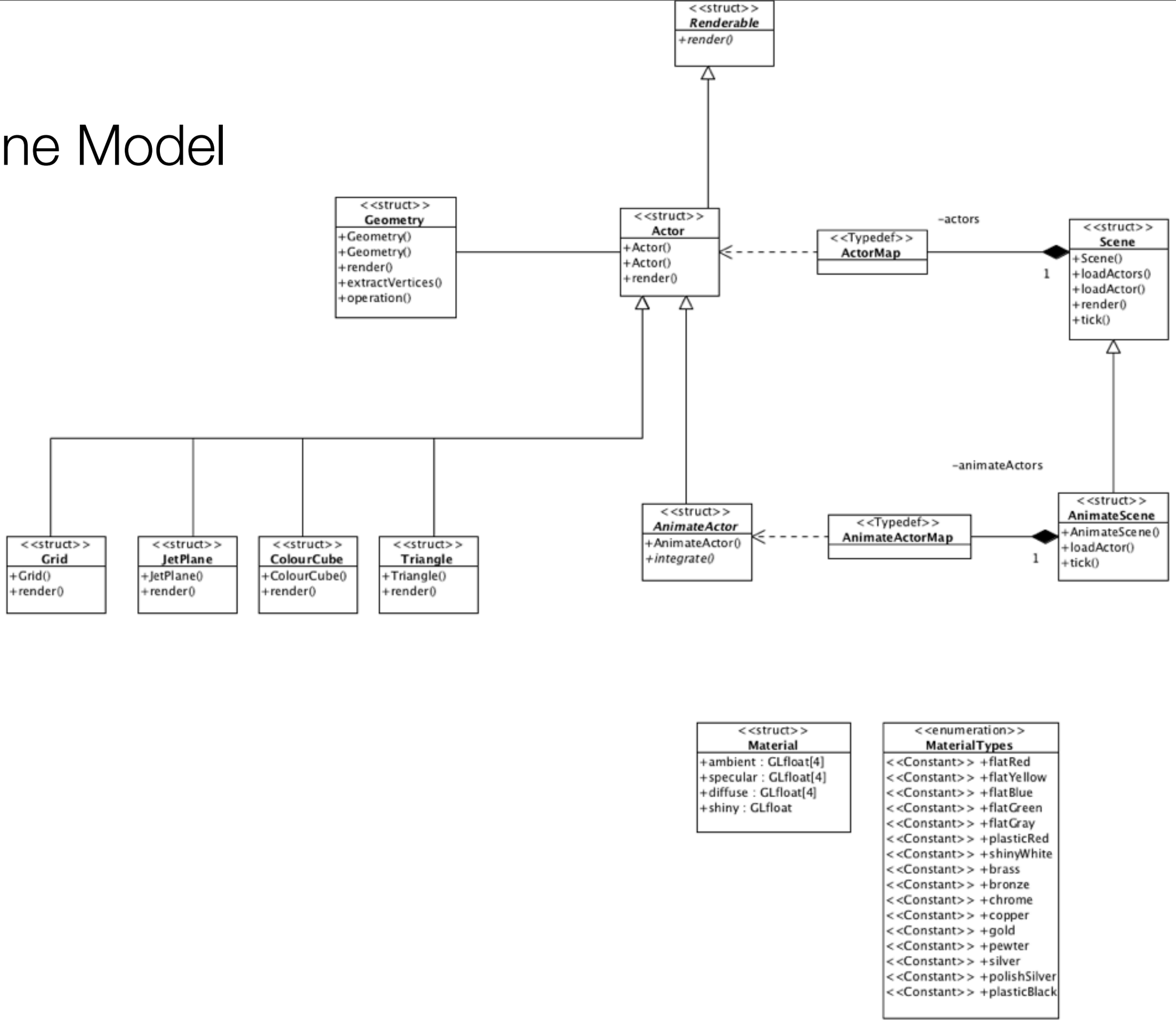
```
struct AnimateScene : public Scene
{
  AnimateScene();
  Actor* loadActor(GeometryMap::value_type &value);

  void tick(float secondsDelta);

  AnimateActorMap         animateActors;
  ForceGeneratorRegistry forceGeneratorRegistry;
};
```

- Overrides loadActor()
- Implement tick()
- Initializes and manages forceGeneratorRegistry

AnimateScene

```
AnimateScene :: AnimateScene()
{
}

Actor* AnimateScene::loadActor(GeometryMap::value_type &value)
{
  Actor *actor = 0;
  if (value.first == "pSphere1")
  {
    PhysicsActor * sphere = new SphereActor(&value.second);
    actor = dynamic_cast<Actor*> (sphere);
    // create an anchor
    Vector3 anchor = sphere->position + Vector3(0,5,0);
    // create force generator
    AnchoredSpringForceGenerator * fg =
    new AnchoredSpringForceGenerator(anchor, 1.0, 3.0);
    // add actor<->forceGenerator pair to registry
    forceGeneratorRegistry.add(sphere, fg);
  }
  else if (value.first == "pSphere2" || value.first == "pSphere3")
  {
    actor = new SphereActor(&value.second);
  }
  else
  {
    return Scene::loadActor(value);
  }
  if (actor)
  {
    animateActors[value.first] = (AnimateActor*) actor;
  }
  return actor;
}

void AnimateScene::tick(float secondsDelta)
{
  forceGeneratorRegistry.applyForce(secondsDelta);

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

Scene Model


<<struct>>
Renderable
+render()
<<struct>>
Geometry
+Geometry()
+Geometry()
+render()
+extractVertices()
+operation()
<<struct>>
Actor
+Actor()
+Actor()
+render()
-actors
<<Typedef>>
ActorMap
1
<<struct>>
Scene
+Scene()
+loadActors()
+loadActor()
+render()
+tick()
-animateActors
<<struct>>
Grid
+Grid()
+render()
<<struct>>
JetPlane
+JetPlane()
+render()
<<struct>>
ColourCube
+ColourCube()
+render()
<<struct>>
Triangle
+Triangle()
+render()
<<struct>>
AnimateActor
+AnimateActor()
+integrate()
<<Typedef>>
AnimateActorMap
1
<<struct>>
AnimateScene
+AnimateScene()
+loadActor()
+tick()
<<struct>>
Material
+ambient : GLfloat[4]
+specular : GLfloat[4]
+diffuse : GLfloat[4]
+shiny : GLfloat
<<enumeration>>
MaterialTypes
<<Constant>> +flatRed
<<Constant>> +flatYellow
<<Constant>> +flatBlue
<<Constant>> +flatGreen
<<Constant>> +flatGray
<<Constant>> +plasticRed
<<Constant>> +shinyWhite
<<Constant>> +brass
<<Constant>> +bronze
<<Constant>> +chrome
<<Constant>> +copper
<<Constant>> +gold
<<Constant>> +pewter
<<Constant>> +silver
<<Constant>> +polishSilver
<<Constant>> +plasticBlack