TAGE – Another Tiny Game Engine

January 2025

by Scott Gordon

Installation *(assumes Java 17 JDK, and JOGL 2.5)*

First, copy items from "`C:\javagaming`" and "`C:\Program Files\Java`" to your computer. Then:

The following items (and full paths to them) need to be added to CLASSPATH environment variable:	The following item needs to be added to the PATH environment variable:

`.`
`jogamp-fat.jar`
`joml-….jar`
`jinput.jar`
`jbullet.jar`
`vecmath.jar`
	`….\jinput\lib`
	(the full path to wherever this folder resides)

Compiling TAGE

Copy the entire contents of the `tage_build` folder into your game folder.
To compile TAGE, use "`clearTAGEclassFiles.bat`", then "`buildTAGE.bat`".
You may use "`buildTAGEjavadocs.bat`" to rebuild the TAGE javadocs.

Setting up a Game Application

Typical game setup is to make a folder for the game application containing these items:

`tage` *(the folder containing the compiled TAGE game engine)*
`assets` *(a folder with the following eight subfolders)*
`-- animations`
`-- defaultAssets`
`-- models`
`-- shaders`
`-- skyboxes`
`-- sounds`
`-- textures`
`myGame` *(folder containing your game's java files – can be renamed)*
✓clearTAGEclassFiles.bat, ✓buildTAGE.bat, ✓buildTAGEjavadocs, ✓compile.bat, ✓run.bat

"HelloDolphin" is a sort of "hello world" example. Use the `assets` folders as follows:

- Place externally-created .OBJ models in the "`models`" folder.
- Place texture image files (e.g., .JPG), and heightmap image files, in the "`textures`" folder.
- Place cubemap image files in the "`skyboxes`" folder (six .JPG files in a subfolder).
- Place sound files (.wav) in the "`sounds`" folder.
- Place exported animated models (.rkm, .rks, and .rka files) in the "`animations`" folder.

The "defaultAssets" folder contains default textures and heightmap used by TAGE and should not be modified.
The "shaders" folder contains .glsl shader programs used by TAGE and can be modified at your own risk!

The TAGE .OBJ importer is from the CSc-155 textbook and has the same limitations:

- Models must be made of triangles (not quads), and must have been UV-unwrapped.
- All three face attributes must be present (vertex, texture coordinates, and normals) – **`f`** #/#/# #/#/# #/#/#
- The model must consist of only a single triangle mesh – no sub-meshes.
- Tags other than v, vt, vn, f are ignored. Material reference is ignored (textures are assigned manually).
- Elements on each line must be separated by exactly one space.

The game's java files go in the "myGame" folder (actually, this folder can be whatever name you want).

The main `.java` file extends `VariableFrameRateGame`, and has the following structure:

```
package myGame;

import tage.*;
import tage.shapes.*;
(other imports are usually also needed – org.joml is almost always needed)

public class MyGame extends VariableFrameRateGame
{
	private static Engine engine;
	public static Engine getEngine() { return engine; }

	private GameObject xxx, xxx, ...	GameObject variable declarations go here
	private ObjShape xxx, xxx, ...		ObjShape variable declarations go here
	private TextureImage xxx, xxx, ...	TextureImage variable declarations go here
	private Light xxx, xxx, ...		Light variable declarations go here
	...
	public MyGame() { super(); }

	public static void main(String[] args)
	{	MyGame game = new MyGame();
		engine = new Engine(game);
		game.initializeSystem();
		game.game_loop();
	}

	@Override
	public void loadShapes()
	{	...		instantiation of shapes go here
	}

	@Override
	public void loadTextures()
	{	...		instantiation and loading of textures go here
	}

	@Override
	public void buildObjects()
	{	...		initial creation of game objects here.
				Each game object typically has a shape and a texture.
	}

	@Override
	public void initializeLights()
	{	...		initialization of global ambient and other lights
	}

	@Override
	public void initializeGame()
	{	...		initialization of other items done here.
				This usually at least includes HUD and camera(s)
	}

	@Override
	public void update()
	{	...		This is automatically called once per frame.
				Therefore, all game logic goes here.
				HUD updates also go here.
	}

	@Override
	public void keyPressed(KeyEvent e)
	{	switch (e.getKeyCode())
		{	...		Desired keyboard assignments go here
		}
		super.keyPressed(e);	ESC already mapped to game exit.
	}				"=" key already mapped to window/fullscreen toggle
}
```

GameObject(s)

A GameObject is an entity (such as a character or building) that can be rendered (displayed) in the game world. GameObjects that are placed in the SceneGraph tree get rendered. Each GameObject holds the following items:

- a "Shape" object (see below).
- a "TextureImage" object (see below)
- a "RenderStates" object (see below)
- matrices for "local" translation, rotation, and scale
- matrices for "world" translation, rotation, and scale
- references to its parent node and children nodes in the scenegraph

It isn't necessary to "add" a GameObject to the SceneGraph. Just instantiate it and it will add itself.

GameObjects handle all object movement and orientation in the game world, via their matrices.

- ✓ GameObjects can be created during the game – but their shapes and textures must be built before the game.
- ✓ multiple GameObjects can utilize the same Shape object.
- ✓ multiple GameObjects can utilize the same TextureImage object.
- ✓ a GameObject's shape and texture can be changed during the game.

There are two "special" GameObjects that are created by the engine:

- a "SkyBox" object the holds the currently active SkyBox shape (if enabled).
- a "root node" object that is at the root of the SceneGraph, and is not rendered.

GameObjects may also hold a HeightMap (if it is used for terrain, generally a "terrain plane").
GameObjects may hold a reference to an associated physics object (if applicable).
It is possible to create an "empty" GameObject (that isn't rendered), and there is a constructor for this purpose.

Shape(s)

Shapes are defined in the class ObjShape, and its subclasses. A "shape" is a vertex geometry and includes its vertices, texture coordinates, normal vectors, and ADS material characteristics. All shapes used in the game must be defined before the game starts running (before the game-loop starts). Shapes cannot be instantiated during game play (although GameObjects *can* be). The following shapes are supported in TAGE:

- `Sphere`
- `Cube`
- `Torus`
- `Plane`
- `Line`
- `Imported Model` (for importing OBJ model files)
- `Manual Object` (for designing a shape – and its vertices – from scratch)
- `SkyBox` (to display 6 separate images in OpenGL cubemap format, without lighting)
- `RoomBox` (a cube built to display a single-image skybox, with lighting)
- `Terrain Plane` (a plane with lots of vertices, intended for terrain height mapping)
- `Animated Object` (for importing .rkm/.rks/.rka animated model files)

TextureImage(s)

A TextureImage is an object that holds a reference to a texture image file (typically `.JPG`), and an integer reference to the associated OpenGL Texture object. All textures used in the game must be read in before the game starts running, and more cannot be instantiated during game play.

It isn't necessary to "add" textures to the render system – just instantiate them and they add themselves.

RenderStates

A `RenderStates` object holds settings for the various ways that a GameObject could be rendered. Most render states are used to tell the engine (and its shaders) which OpenGL settings to use when rendering the object. The following render states are supported in TAGE, and are set separately for each GameObject:

- Enable or Disable rendering
- Apply lighting (or not)
- Enable or Disable depth testing (such as for a skybox)
- render in wireframe
- use OpenGL texture tiling (none, repeat, mirrored-repeat, or clamp-to-edge), and how many
- render in a solid color
- render hidden faces (such as if the camera can go inside the object)
- apply environment mapping (to make a "chrome" looking object)
- specify primitive type (default is triangles, but set to line if the Shape is line)
- apply a pre-rendered rotation to compensate for a model that doesn't properly face forwards
- transparency (not yet implemented)

Light(s)

A game can instantiate an unlimited number of Light objects. If added via the SceneGraph, they become active. They can be turned on and off (enabled/disabled), or added/removed. TAGE supports three types of lights:

- Global ambient (only one)
- Positional lights
- Spotlights

The game application can specify ADS coefficients separately for each light.
Distance attenuation can also be specified (the default settings effectively disable distance attenuation).

The management of the list of active lights is handled by a LightManager object.
The game application doesn't interact with LightManager directly, instead functions in SceneGraph are used.

Camera(s)

A Camera is an object that is used as a viewpoint for rendering. There is always at least one camera.
Each camera has its own *location* and *orientation*.
A camera's *location* is a point in space, and is defined as a Vector3f.
A camera's *orientation* is defined with three vectors U, V, N – corresponding to "right", "up", and "forward".

Each Viewport automatically has a camera assigned to it.
The default viewport is named "MAIN", and therefore to get a reference to the default camera, use:

```
engine.getRenderSystem().getViewport("MAIN").getCamera()
```

Viewport(s)

TAGE supports an unlimited number of Viewports.
If no viewports are specified, a single Viewport named "MAIN" is created.
Each viewport has its own active camera for its rendering viewpoint.
The game application may specify colored borders for each Viewport, if desired.

HUD (Heads-Up-Display)

TAGE's HUD Manager supports two simple HUD strings that can be placed anywhere on the display.
The HUDs are implemented as GLUT strings and are the only part of TAGE that uses deprecated OpenGL.

Engine

There is an "Engine" class which is the topmost driver in the TAGE structure. It is the first thing that the game application should instantiate. The Engine instantiates (and can be used to acquire) the following TAGE tools:

- RenderSystem (manages viewports and all low-level rendering of objects, and the game loop)
- SceneGraph (manages hierarchical object definition, and used for adding lights, node controllers, etc.)
- Input Manager (for accepting input from a gamepad and other devices)
- HUD Manager
- Light Manager
- Audio Manager

The Engine class holds a static reference to the current Engine object.
So the engine can always be retrieved from the Engine *class* itself.

Node Controller(s)

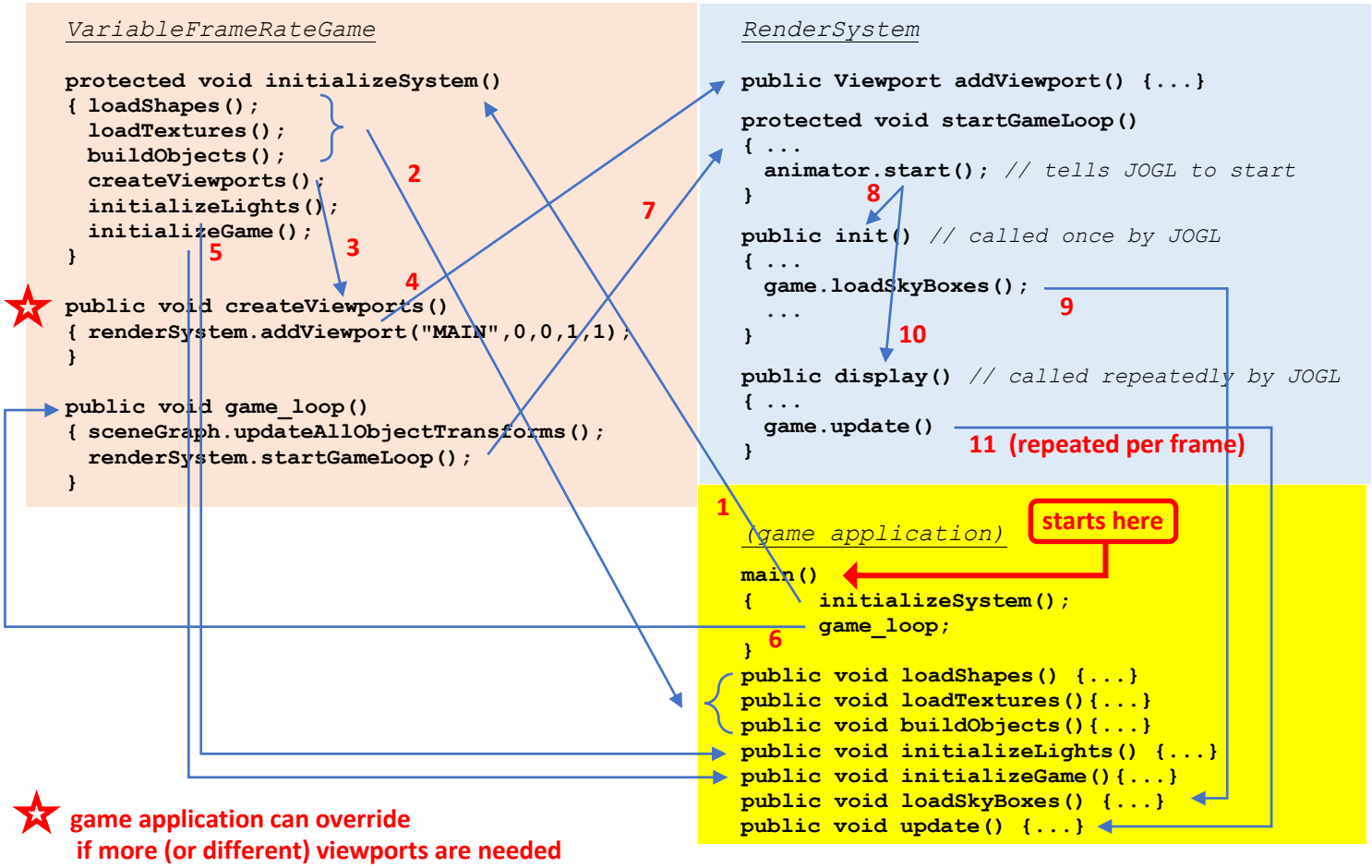
TAGE has one built-in node controller – a simple rotation controller that causes a GameObject to spin.
The NodeController class can be extended to create custom node controllers, to do this the custom controller class will need to override the `apply()` function. Then, to use a node controller in the game:

1. instantiate the node controller
2. add one or more target GameObject(s)
3. add the node controller to the game via the accessor in SceneGraph

The controller(s) can then be enabled or disabled when desired.
When a GameObject is removed, any NodeController references to it are automatically removed.

VariableFrameRateGame

TAGE comes with this one predefined game loop, which your game application must extend. It is useful to understand what this does, so that you know when your override functions get called, and by whom:



Functions related to GameObjects , Shapes , and Textures

- *build a game object*
  - > instantiate a **`GameObject`** in the game's buildObjects() section.
  - > use one of the constructors in GameObject, specifying the shape and texture.
  - > for example: moon = new GameObject(GameObject.root(), mySphere, moonTexture);
  - > can also instantiate new GameObjects in update() during game play.
- *build a shape*
  - > instantiate the **`ObjShape`** directly in the game's loadShapes() section.
  - > all shapes must be instantiated before the game starts.
  - > for example: mySphere = new Sphere();
  - > a shape can be simple built in (sphere, cube, etc.), or an imported model, etc.
- *load a texture*
  - > instantiate a **`TextureImage`** in the game's loadTextures() section.
  - > place texture image files (e.g. JPG) in the "textures" asset folder.
  - > all textures must be loaded from image files before the game starts.
  - > for example: moonTexture = new Texture("moonImage.jpg");
  - > to tile a texture, get the object's RenderStates, and use the tiling functions there
- *place or move an object*
  - > modify local matrix transforms (*local translation, local rotation, local scale*)
  - > these matrices are in GameObject, and accessor functions are there.
  - > for example: moon.setLocalTranslation((new Matrix4f()).translation(1,1,0));
  - > can also set the location of an object directly using the setLocalLocation() function.
  - > there are convenience functions, such as making an object look at another object.
  - > matrix transformations are always relative to the object's parent.
- *look up object location*
  - > access world matrix transforms (*world translation, world rotation, world scale*)
  - > for example: Vector3f loc = moon.getWorldLocation();
  - > there are also accessors for world *up*, *right*, and *forward* orientation vectors.
- *get an object's vertices*
  - > vertices and accessors to them are in the ObjShape associated with the object.
- *--- texture coordinates*
  - > texture coordinates (and accessors) are in the ObjShape associated with the object.
- *--- normal vectors*
  - > normal vectors (and accessors) are in the ObjShape associated with the object.
- *get/set object's material*
  - > accessors to material ADS characteristics and shininess are in the ObjShape.
  - > don't confuse *texture* and *material*. Texture is an image, material is for lighting.
- *remove a game object*
  - > use the removeGameObject() function in SceneGraph.
  - > get the SceneGraph object using the getter in Engine: sg = engine.getSceneGraph();
  - > for example: sg.removeGameObject(moon);
  - > if the object has children, they must be removed first, or TAGE displays an error.
- *set an object's parent (for hierarchical objects)*
  - > get a reference to the desired parent, then use setParent() function in GameObject.
  - > TAGE will update the child lists in both the new parent and the previous parent.
  - > for example: moon.setParent(sun);
  - > can also get an iterator for a GameObject's children.
- *change the appearance of an object*
  - > start by getting the **`RenderStates`** object, e.g.: rs = moon.getRenderStates();
  - > the RenderStates object has numerous functions to change an object's appearance:
    - ✓ *enable/disable rendering the object*
    - ✓ *enable/disable whether lighting is applied to the object*
    - ✓ *environment mapping*
    - ✓ *wireframe*
    - ✓ *transparency (not yet implemented)*
    - ✓ *render hidden surfaces (useful for seeing the interior)*
    - ✓ *tiling the object's texture*
    - ✓ *render in a specified solid color*
    - ✓ *set a rotation to correct an incorrect model orientation*

Functions related to Lighting

• *set global ambient light*
> use the static function in the **`Light`** class.
> set the RGB characteristic, for example: Light.setGlobalAmbient(0.5f, 0.5f, 0.5f);

• *add a light*
> individual lights are either *positional* or *spotlight*. Positional is the most common.
> initial lights are defined in the initializeLights() section.
> more lights can be added while the game is in progress, in update().
> step 1 – instantiate the Light object
> step 2 – set the light's intended location using setLocation()
> step 3 – add the light to the scene using the addLight() function in Scenegraph.
> for example:

```
myLight = new Light();
myLight.setLocation(new Vector3f(5.0f, 4.0f, 2.0f));
(engine.getSceneGraph()).addLight(myLight);
```

• *delete a light*
> use the removeLight() function in SceneGraph.
> for example: (engine.getSceneGraph()).removeLight(myLight);

• *set light characteristics*
> setType() sets the light to positional (the default) or spot.
> enable(), disable(), and toggle() turn the light on and off.
> setLocation() places the light at a specified world location.
> setAmbient(), setDiffuse(), and setSpecular() set ADS values (in RGB) for this light.
> setDirection(), setCutoffAngle(), etc. apply only to spotlights.
> there are also functions for setting distance attenuation and range.

• *set material values*
> the effects of light on an object is partly determined by the object's material.
> accessors to material ADS characteristics and shininess are in the ObjShape.

• *get normal vectors*
> the effects of light on an object is partly determined by the object's normal vectors.
> accessors to an object's normal vectors are in the object's ObjShape.

• *disable lighting on a particular object*
> an object can be made to have color or texture but be unaffected by lighting.
> there are functions to enable/disable lighting on an object, in RenderStates.

• *get the number of lights*
> first, get the **`LightManager`** using the Engine: lm = engine.getLightManager();
> then, use the getNumLights() function. For example: nl = lm.getNumLights();
> it is unlikely that a game application will need to use the LightManager.

Functions related to Cameras

• *get the camera object*
> cameras are associated with viewports. By default there is one "main" viewport.
> to get the camera, first get the RenderSystem, then use that to get main viewport.
> for example:

```
rs = engine.getRenderSystem();
mainViewport = rs.getViewport("MAIN");
cam = mainViewport.getCamera();
```

• *move the camera*
> cameras have location and orientation. To move it, use setLocation().
> for example: (mainViewport.getCamera()).setLocation(new Vector3f(0,3,2));

• *turn the camera*
> cameras orientation is specified with three vectors U, V, and N.
> the three vectors are a left-handed system and must remain orthogonal.
> set (or get) the vectors using the accessors in Camera.
> for example: (mainViewport.getCamera()).setN(new Vector3f(0,0,-1));

• *have camera "look at"*
> there are functions for making the camera look at an object or specific location.
> for example: (mainViewport.getCamera()).lookAt(moon);

Functions related to HUDs

- *get the HUD manager* > use the accessor in Engine: `hm = engine.getHUDmanager();`
- *set text, loc, color, font* > there are two HUDs that can be customized using the HUD manager.
  > for example: `hm.setHUD1("you win!", new Vector3f(1,0,0), 15, 15);`

Functions related to the SceneGraph

- *get the SceneGraph* > the **`SceneGraph`** is a tree data structure that holds the GameObjects.
  > it can be used to build hierarchical objects or hierarchical systems of objects.
  > use the accessor in Engine to get the SceneGraph: `sg = engine.getSceneGraph();`
- *get the root node* > the root node is a special GameObject that is not rendered. It has location (0,0,0).
  > use the accessor in GameObject: `root = GameObject.root();`
- *add a GameObject to the scene graph* > simply use one of the constructors – it is added to the scene graph automatically.
  > for example: `moon = new GameObject(GameObject.root(), mySphere, moonTexture);`
  > the first parameter is the parent object (usually the root node).
- *remove a GameObject* > use the `removeGameObject()` function in SceneGraph.
  > for example: `sg.removeGameObject(moon);`
  > if the object has children, they must be removed first, or TAGE displays an error.
  > if the object has node controller(s), TAGE will remove those references.
- *set an object's parent (for hierarchical objects)* > get a reference to the desired parent, then use `setParent()` function in GameObject.
  > TAGE will update the child lists in both the new parent and the previous parent.
  > for example: `moon.setParent(sun);`
  > can also get an iterator for a GameObject's children.

Functions related to SkyBoxes

- *load skybox from file(s)* > TAGE uses OpenGL cube maps for drawing skyboxes.
  > all skyboxes must be loaded from image files before the game starts.
  > the six images are placed in a folder, which is put in the "skyboxes" asset folder.
  > in the game application, override `loadSkyBoxes()`
  > call the `loadCubeMap()` function in SceneGraph, then make it the active skybox.
  > for example: `fluffy = (engine.getSceneGraph()).loadCubeMap("clouds");`
  > it is also necessary to make it the active skybox, and enable it (see below).
- *set the active skybox* > multiple skyboxes can be loaded, but only one skybox is active at a given time.
  > to set which skybox is active, use `setActiveSkyboxTexture()` in SceneGraph.
  > for example: `(engine.getSceneGraph()).setActiveSkyBoxTexture(fluffy);`
- *enable the skybox* > if skybox is enabled, the active skybox is rendered.
  > to enable (or disable) the skybox, use `setSkyBoxEnabled()` in SceneGraph.
  > for example: `(engine.getSceneGraph()).setSkyBoxEnabled(true);`

Functions related to Terrain

- *make a terrain object* > instantiate a **`TerrainPlane`** shape, then make a GameObject with this shape.
- *load a height map* > height maps are stored in image files (e.g., JPG). Place it in the "textures" asset.
  > load the image file into a TexureImage, like any other texture.
  > assign the height map texture to the GameObject using `setHeightMap()`
  > for example: `ground.setHeightMap(hills);`
- *get height at location* > use the `getHeight()` function in GameObject: `height = ground.getHeight(125, 32);`

Functions related to Viewports (and the display)

- *get the RenderSystem* > use the accessor in Engine: `rs = engine.getRenderSystem();`
- *set window dimensions* > use the function in RenderSystem: `rs.setWindowDimensions(1900,1000);`
- *set window title* > use the function in RenderSystem: `rs.setTitle("robot game");`
- *toggle fullscreen mode* > the functions are in RenderSystem, but they are not fully implemented.
- *add a viewport* > use the function in RenderSystem: `rs.addViewport("rightVP", .75f, 0, .25f, .25f);`
  > all viewports have a string name, specified in the first parameter of addViewport().
  > TAGE creates one viewport automatically, named "MAIN".
- *get a viewport by name* > use the function in RenderSystem: `mainVP = rs.getViewport("MAIN");`
  > this is most commonly used to get the camera associated with the main viewport.
- *get a viewport's camera* > use the function in the **`Viewport`** class: `cam = mainVP.getCamera();`
- *set viewport loc & size* > these are specified at the time of viewport creation, relative to the window.
  > in the above example, lower left viewport position is 75% of X-axis, 0% of Y-axis.
  > in the above example, the size of the viewport is 25% of (window) X, 25% of Y
- *draw viewport border* > a border can be enabled/disabled, and the width and color can be set.
  > for example:

```
rightVP.setHasBorder(true);
rightVP.setBorderWidth(4);
rightVP.setBorderColor(0.0f, 1.0f, 0.0f);
```

Functions related to the Physics Engine

- *get the PhysicsEngine* > use the accessor in SceneGraph: `pe = (engine.getSceneGraph()).getPhysicsEngine();`
- *set gravitational force* > gravitational force is specified by a direction and magnitude, in a 1x3 float array.
  > gravity is typically set in the world -Y direction (but this is not required).
  > for example:

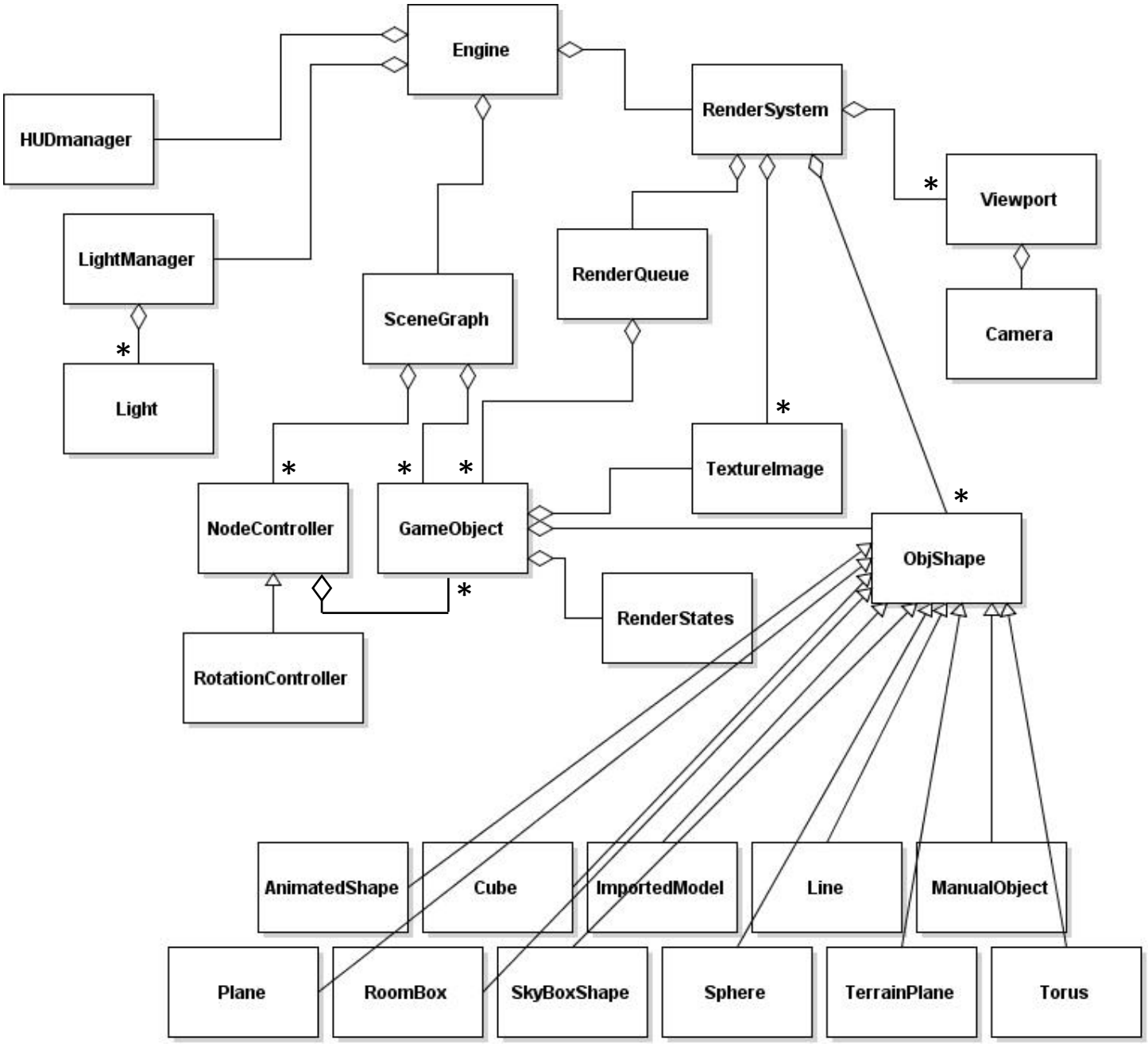
```
float[] gravity = {0f, -5f, 0f};
pe.setGravity(gravity);
```

- *add physics object* > available physics shapes are *box, sphere, cone, capsule, cylinder,* and *static plane.*
  > functions for instantiating a physics object in the physics world are in SceneGraph.
  > for example: `ballP = sg.addPhysicsSphere(mass, ballLocation, 0.75f);`
  > there are numerous parameters that can be set, e.g.: `ballP.setBounciness(1.0f);`
- *associate physics object with graphics object* > this is done on the GameObject side.
  > for example: `ball.setPhysicsObject(ballP);`
  > this does not cause a graphic object to automatically move with its physics object.
- *look up physics object for a graphics object* > to move a graphics object by physics, look up the corresponding physics object
  > use the accessor in GameObject: `bp = ball.getPhysicsObject()`
- *move a graphics object based on physics object* > physics objects have a single transform matrix, containing rotation and translation.
  > copy the physics translation into the GameObject's localTranslation matrix.
  > copy the physics rotation into the GameObject's localRotation matrix.
- *delete a physics object* > use the function in SceneGraph: `sg.removePhysicsObject(ballP);`
  > TAGE removes the renderable, and the association with the graphics object.
- *render physics world* > use the function in Engine: `engine.enablePhysicsWorldRender()`
  > can also enable/disable rendering the graphics world (this is rare).

Process for creating a Physics World

- The SceneGraph holds a reference to the physics engine. Use the physics engine to set gravity.
- Instantiate and add PhysicsObjects to the physics world using the methods in the SceneGraph class.
- Associate GameObjects with PhysicsObjects using the `setPhysicsObject()` function in GameObject.
- Get a GameObject's associated physics object using the `getPhysicsObject()` function in GameObject.
- There are no reverse references from a PhysicsObject to the associated GameObject.
- To operate the physics, call the physics engine `update()` function, from the game's `update()`.
- You can view the physics world by using the TAGE engine's `enablePhysicsWorldRender()`.
- The physics objects appear in wireframe, and are visible even if behind other objects.
- There are similar functions for disabling physics world rendering, and enabling/disabling graphics rendering.

TAGE UML Class Diagram



TAGE Extension Packages

Input Package

This is a wrapper for Jinput, written by John Clevenger and Russell Bolles.
Jinput is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.
This enables associating an action with a keyboard or gamepad component (or all keyboards, gamepads).
See example #02a and #02b.

Networking Package

This is a set of tools for setting up server-client communication, written by Kyle Matz.
See example #08a.

Physics Package

This is a wrapper for JBullet, written by Russell Bolles and Kyle Matz.
JBullet is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.
Physics objects should be constructed (and removed) using the methods in the SceneGraph class.
See example #13a.

Audio Package

This is a wrapper for JOAL, written by Kenneth Barnett, based on work by Mike McShaffry.
JOAL is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.
See example #15a.

AI Package

This is an implementation of Behavior Trees by Kenneth Barnett, for controlling an NPC.
See example #14a.

Animation Support

This is a port of the RAGE exporter, renderer, and associated shaders developed by Luis Gutierrez.
It enables importing and rendering models that have associated animations.
The model and its associated skeleton and animations must be exported from Blender using the RAGE exporter, which is installed in Blender as a set of three plugin scripts.
See example #12a.

Animated models must have been exported using the Blender exporter scripts.
These are an `.rkm` (model file) and `.rks` (skeleton file), and an `.rka` file for each animation.

OpenGL and related rendering functions

- to get the GLCanvas - use the function in RenderSystem
- display(), init(), dispose(), and reshape() are found in RenderSystem.
- enable/disable physics world visualization rendering - use the setter in Engine.
- enable/disable normal graphics rendering - use the setter in Engine.

Math libraries

The vast majority of TAGE uses the **JOML** math library, especially classes Matrix4f, Vector3f, and Vector4f.
These are used for specifying locations, directions, transforms, and even RGB colors.
JOML is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.

The physics wrapper (which uses JBullet) uses the **vecmath** math library.
Vecmath is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.

The animation renderer uses the **RML** math library, which was developed by Ray Rivera.
RML is built into TAGE, and is a subfolder in TAGE, so a separate classpath entry is not needed.