TAGE - Tiny Game Engine

by Scott Gordon

Installation *(assumes Java 11 and JOGL 2.4)*

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:

`.`
`jogl-all.jar`
`gluegen-rt.jar`
`joal.jar`
`joml-….jar`
`jinput.jar`
`jbullet.jar`
`vecmath.jar`
	`….\jinput\lib`
	(the full path to wherever it resides)

Setting up a Game Application

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

`tage` *(a folder containing the TAGE game engine)*
`assets` *(a folder with the following eight subfolders)*
`-- animations`
`-- defaultAssets`
`-- models`
`-- scripts`
`-- shaders`
`-- skyboxes`
`-- sounds`
`-- textures`
`myGame` *(folder containing your game's java files)*
`compile.bat & run.bat`

"HelloDolphin" is a sort of "hello world" example. In general, use 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 javascript (.js) files in the "scripts" folder.
- Place cubemap image files in the "skyboxes" folder (e.g., 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 should not be modified.

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, and f are ignored. Material reference is ignored (textures assigned manually).
- Elements on each line must be separated by exactly one space.

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

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 initializeGame()
	{	...		initialization of other items done here.
			This usually at least includes lights 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
}
```

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. 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.

GameObject(s)

A GameObject is an entity (such as a character or building) to be rendered (displayed) in the game world. More specifically, a GameObject is a node in the SceneGraph. Each GameObject holds the following items:

- a "Shape" object (see above)
- a "TextureImage" object (see above)
- 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 scene graph. Just instantiate it and it will add itself.

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

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
- do OpenGL texture tiling (none, repeat, mirrored-repeat, or clamp-to-edge)
- 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 rotation to compensate for a model that doesn't properly face forwards
- transparency (not yet implemented)

Camera(s)

A Camera is an object that is used as a viewpoint for rendering.
It has a location and an orientation.
The location is defined as a Vector3f.
The orientation is defined with three vectors U, V, N – those are right, up, and forward respectively.

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()
```

Light(s)

A game can instantiate and add an unlimited number of lights.
TAGE supports three types of lights:

- Global ambient
- Positional lights
- Spotlights

The application can specify ADS coefficients separately for each light.
Distance attenuation can also be specified, but the default settings effectively disable it.

The management of the list of lights is handled by a LightManager object.
The game application usually doesn't need to interact with the LightManager directly.

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.

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.

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

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)

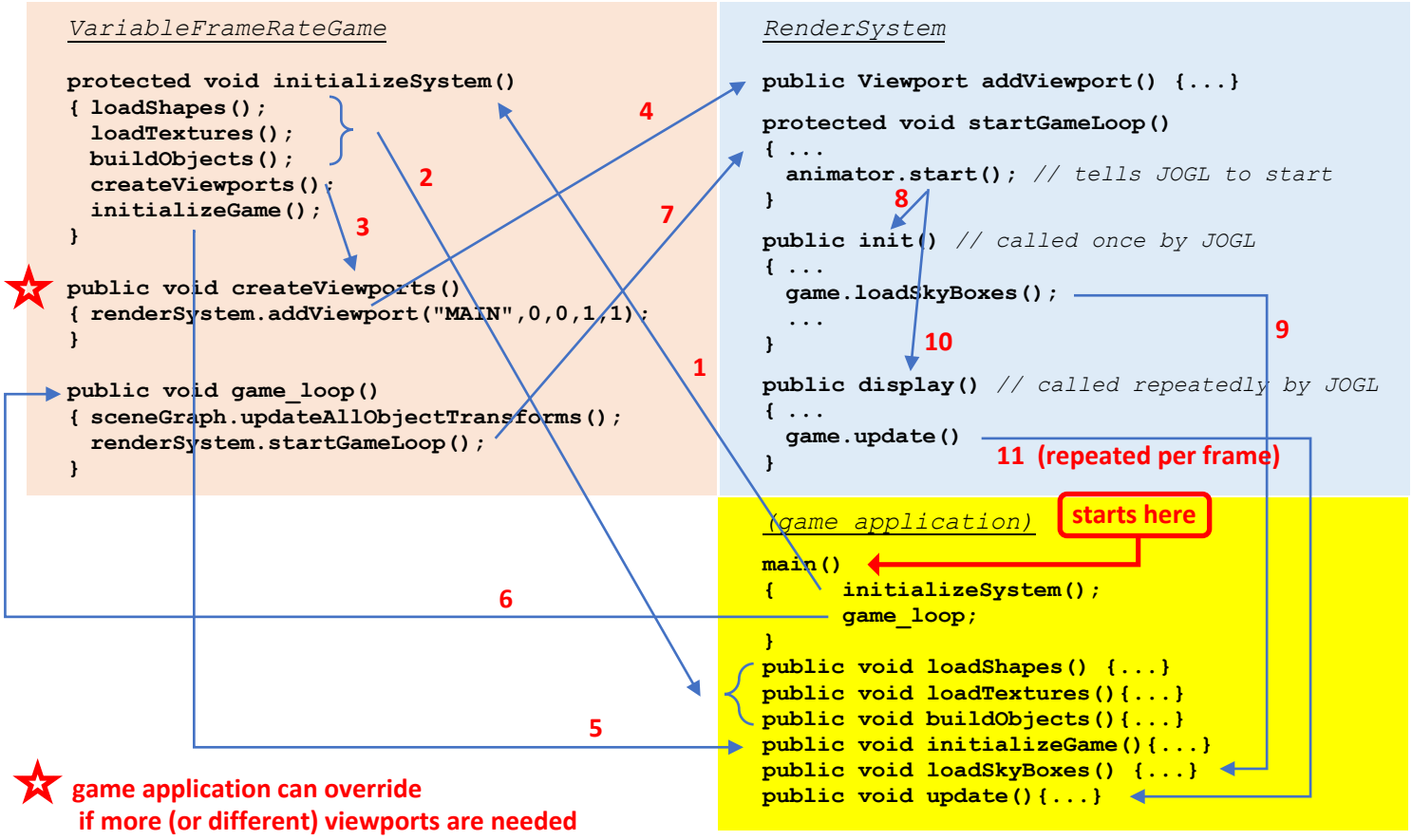
TAGE has one built-in node controller – a simple rotation controller that causes a GameObject to spin.
The NodeController class can be extended to create custom node controllers (see HelloDolphin for an example), the game application will need to override the `apply()` function. To use a node controller:

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 (see HelloDolphin for examples).

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:



Summary of Basic Functions

add a shape

instantiate the shape directly in the game's `loadShapes()` function:

```
new Torus
new Sphere
new Cube
new ImportedModel(filename)
```

etc.

add a texture

instatiate the TextureImage directly in the game's `loadTextures()` function:

```
myTexture = new TextureImage(filename)
```

build an object

instantiate `GameObject` in the game's `buildObjects()` function:

```
myGameObject = new GameObject(GameObject.root(), shape, texture)
```

can also instantiate new GameObjects in `Update()` during game play.

add a light

instantiate `Light`, then call `setLocation()`, then add it using Scenegraph:

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

set global ambient light

`setGlobalAmbient()` is a static function in the Light class.

Orient the camera

get the camera from the viewport, then use `setLocation()`, `setU()`, etc.

```
RenderSystem rs = engine.getRenderSystem();
thisViewport = rs.getViewport("MAIN");
thisViewport.getCamera()).setN(new Vector3f(0,0,-1));
```

There are also lookAt() functions to point it in a direction or at a particular object.

Move an object around

`setLocalTranslation(), setLocalRotation(), setLocalScale()`

All of these functions are in `GameObject`. They expect a Matrix4f.

e.g.: `myObj.setLocalTranslation(new Matrix4f()).translation(1,1,0);`

There are also convenience functions for setting the location of an object directly.

Add a SkyBox

Override `loadSkyBoxes()`.

In it, call loadCubeMap() and specify a folder name that contains the 6 JPG files.

Then set it as the active texture, and enable the skybox. e.g.:

```
fluffy = (engine.getSceneGraph()).loadCubeMap("clouds");
(engine.getSceneGraph()).setActiveSkyBoxTexture(fluffy);
(engine.getSceneGraph()).setSkyBoxEnabled(true);
```

All cubemaps must be loaded (with loadCubeMap()) once, in loadSkyBoxes().

But the active skybox can be changed during the game, or enabled/disabled.

Change a node's parent
(for hierarchical objects)

`setParent()` function is in `GameObject`.

Get a reference to the desired parent object, and call `setParent()`.

TAGE will set the node's parent, remove it from the old parent's child list, add it to the new parent's child list.

e.g.: `myNode.setParent(desiredParent);`

Make multiple viewports

override CreateViewports(), and call addViewport() for each viewport desired.

`addViewport()` is in the RenderSystem.

The parameters are name, bottom, left, width, height. e.g.:

`renderSystem.addViewport("MAIN",0,0,1,1);`

You can also specify a colored border for a Viewport.

Overview of other basic Functions by Class

Functions in ObjShape: set material & shininess characteristics for this shape

Functions in GameObject: set/get local & world transforms and their propagation properties
set/get associated Shape and TextureImage
set/get whether it is used for terrain
set/get this object's height map, if it has one
set/get this object's scenegraph parent node and children nodes
look-at(), for pointing this object at a location or another object
set/get this object's associated physics object, if applicable
get the height of this object's height map at a particular world point, if applicable.

Functions in RenderStates: enable/disable rendering this object
render this object with/without lighting
render this object with a solid color, and specify the color
render this object in wireframe
render both the inside and outside of this object
render this object with environment mapping
make the object transparent (not yet implemented)

Functions in RenderSystem: set the title at the top of the window
add/get one of the viewports (by name)
get the height of a texture map at a particular texture coordinate
(usually the related function in GameObject is more useful)

Functions in SceneGraph: add a Light to the game
add a node controller
remove a GameObject from the scenegraph
get the number of game objects
get the root game object (root is the topmost "dummy" node in the scenegraph)
set the currently active skybox cubemap, and enable or disable the skybox.

Functions in Camera: set the location and orientation
look-at() – aims camera at a location or at a specified object

Functions in Light: set ADS characteristics for global ambient light (as static functions)
set type of light (positional or spotlight)
set ADS characteristics for this light
set location
set direction, cutoff angle, and offAxisExponent (if spotlight)
set constant, linear, and quadratic distance attenuation factors
enable/disable this light
set the range for this light (not yet implemented)

Functions in HUDmanager: set HUD string(s) for hud #1 and/or hud #2
set location of hud string(s)

Functions in NodeController: enable/disable (or toggle whether it is enabled/disabled)
add a "target" object for it to control
get the amount of time this controller has been most recently enabled

Functions in Engine: getEngine() is a static method that returns the currently active engine object
getRenderSystem(), getSceneGraph(), getHUDmanager(), getLightManager()
getInputManager() – this is useful for controlling a gamePad.

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.
RML is built into TAGE, and is a subfolder in TAGE, so a separate classpath entry is not needed.

TAGE Extension Packages

Input Package

This is a wrapper for Jinput, written by John Clevenger.
Jinput is a separate item in the classpath, as it is a separate JAR file outside of the TAGE folder.
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.
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

TAGE can import and render 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.
The exporter and playback renderer and shader was developed by Luis Gutierrez.
See example #12a.

TAGE UML Class Diagram