GameObject

This is the abstract class used for every single thing in the game. If it needs to run logic, collide with something, or be drawn to the screen, it’s a GameObject. All other types of objects should be child classes of GameObject, or else children of children of GameObject.

All GameObjects have the following attributes:

* x, y – pair of floats indicating the object’s position, mapped from the upper left. If the object does not need to be represented on screen or do collision checking, just set these to 0.
* collisionX, collisionY – pair of floats indicating the object’s width and height, respectively. If the object does not need to do collision checking, set these to 0.
* visible – Boolean value indicating whether the object is currently visible (that is, in the flashlight beam). Set it to false at the end of the update() function for most objects.
* opaque – Boolean value indicating whether the object stops the flashlight beam. For example, walls have it set true, enemies have it set false.
* fogVisible – Boolean value indicating whether the object is visible once seen if not currently in the flashlight beam.
* beenSeen – Boolean value indicating whether the object has previously been seen. Not particularly valuable unless fogVisible has also been set true.
* collides – Boolean value indicating whether the object has collision logic. If set to false, the object will not be included in the collision graph.
* removed – Boolean value indicating whether the object has been flagged for removal. Objects with destroyed set to true will be removed from the game at the end of the current game loop.
* ui – Boolean value indicating whether the object is part of the UI or not (and thus, whether it should move when the camera does)

All GameObjects have the following functions:

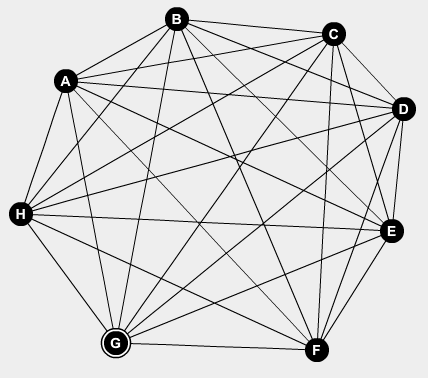
* abstract void collide(GameObject other) – This function takes in another GameObject and runs whatever code this one needs if it collides with the other. Use the lists to determine what type of object it is, and then run appropriate code. Or no code, if these two objects don’t interact with each other. This should only be called from the global collisionCheck() function
* abstract void update() – This function runs the object’s logic: updates its position, runs AI scripts etc.
* abstract void render() – This function draws the object to the screen, if it is close enough to the window to be visible.

For each type of GameObject, we must also create a pair of global functions: one which creates the object (new), adds it to the requisite lists, and then calls generateCollisionMap() (see below), and one which removes the object from the lists which contain it.

The Collision Map

Instead of having each object check collision against every other object, we will create and maintain a map of all objects with collision.

Consider for an example a scenario where you have 8 objects with collision. If every object checks collision against every other object, you have 56 collision checks. On the other hand, consider the following fully connected graph, with the eight objects as the vertices:



This graph has 28 edges. If we check collision along these edges, we will cover every possible collision, while using about half as many collision checks as checking every object against every other object. In order to do this, we will create a class called Collider.

EXCEPTION: Collisions related to light sources are not part of this process, and are calculated separately, because otherwise you would need to remake the collision map essentially every frame, which would utterly defeat the computational advantages of this method.

Collider

This class is used to check collisions. It is *not* a GameObject, as it is used only for pure collision logic purposes. It contains the following attributes:

* object1, object2 – a pair of GameObjects, which both have *collides* set to true

It contains the following function:

* void collide() – This function compares object1 and object2 and, if they are in collision, calls each object’s collide() function, with the other object passed as the parameter.

void generateCollisionMap()

This global function uses the list of all objects, and generates a list of Colliders. This is the collision map. This function should be called *once, at the end of the game loop*.

Game Loop

The game loop takes the following shape:

Draw the Background

Set mapChanged to false

For all Colliders:

collide();

For all LightSources:

createRays();

render();

For all Objects:

update();

For all Objects (Except LightSources):

render();

cleanObjects(); //removes all objects that have been flagged for removal

For all LightSources:

cleanRays(); //removes all rays and tracers, resetting for the next loop

If mapChanged has been set true:

generateCollisionMap();

Object Lists

All GameObjects are stored in ArrayLists of the relevant type. Even if there is only one GameObject, it MUST be contained in a list. There is one master list, named *objects*, which is an ArrayList of ArrayLists, pointing to each of the other ArrayLists.