Game Engine

Messaging System

Most interactions between components of the engine consist of passing messages to and fro, so before I talk about the actual game engine, I need to talk about the messaging system that the engine uses.

There are two main types of objects in the messaging system: dispatchers and listeners. Dispatchers, as their name implies, send out messages; listeners, on the other hand, listen to message events and respond accordingly. Listeners *subscribe* to dispatchers for specific messages; dispatchers then keep a list of these subscriptions. When a dispatcher sends out a message, what it actually does is go through its list of subscribers, check which listeners listen to the type of message it's about to send, and then calls a *callback* function for those listeners with the broadcasted message as an argument.

Let's see an example of this. Below, a Level object broadcasts an *update* message, which tells other components of the engine that the game has advanced one frame. Here is the method that performs this broadcast:

```
public void update() {
	//update physics
	this.updatePhysics();

	//check collisions
	this.checkCollisions();

	//broadcast update message
	for (Controller c : this.ctrl_list) {
		c.onMessage(this.update_msg);
	}

	//send the message to all the other listeners
	this.broadcast(this.update_msg);
}
```

Ignore *updatePhysics* and *checkCollisions* for now; we're interested in the last half of the code. Here, the Level object sends the update message for all the entity controllers (I'll explain this later), and then broadcasts the message for the rest of its listeners who are subscribed to update messages. *broadcast()* basically does the same thing as the *for* loop – that is, it calls the *onMessage* method (i.e., the callback function) of the listeners – except it doesn't call *onMessage* for all listeners, just select ones.

What each listener does with the update message depends on the listener; for example, here is an example of the controller of the player entity doing something with the update message:

```
	public void onMessage(Message msg) {
		EntityMoveMessage move_msg;
		switch (msg.getType()) {
		case LEVEL_UPDATE:
			move_msg = new EntityMoveMessage(MsgType.ENTITY_COMMAND_MOVE,
			this.player, -5.0, 0.0, 0.0);
			this.player.onMessage(move_msg);
			break;
		default:
			break;
		}
	}
```

As you can see above, once the player controller receives the update message, it then sends a command message to the player entity to move. This wouldn't happen in the actual game, of course; this is just an example.

You might be wondering: how does the player entity know how to move once it gets the command message from the controller? This is where the messaging system shines: *you can send any kind of message with any kind of data using the system*. All you have to do is to subclass the Message base class and include more data in the subclass. For example, the EntityMoveMessage class looks like this:

```
public class EntityMoveMessage extends EntityMessage {
	private double x, y, rot;
	public EntityMoveMessage(MsgType type, Entity e, double x,
	double y, double rot) {
		super(type, e);
		if ((type != MsgType.ENTITY_MOVE) &&
		(type != MsgType.ENTITY_COMMAND_MOVE)) {
			throw new IllegalArgumentException("Message type must be
			ENTITY_MOVE or ENTITY_COMMAND_MOVE");
		}
		this.x = x;
		this.y = y;
		this.rot = rot;
	}

	//only instantiate getters, not setters
	//we don't want the message values changed once it's been sent
	public double getX() { return this.x; }

	public double getY() { return this.y; }

	public double getRotation() { return this.rot; }
}
```

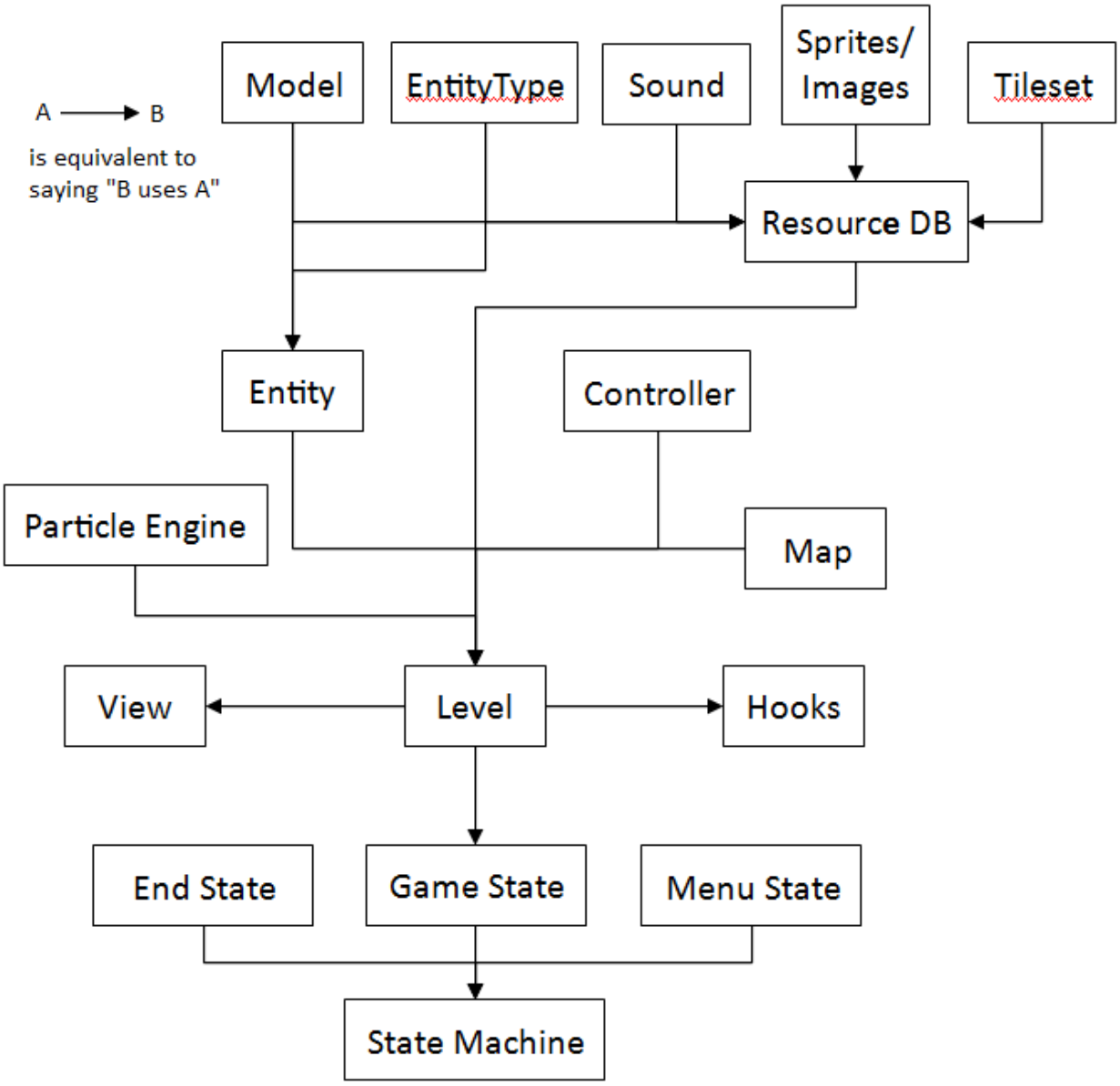
The controller above just has to fill out *x, y*, and *rot*, all that's all the player entity needs to know to move accordingly, as shown below:

```
public void onMessage(Message msg) {
	switch(msg.getType()) {
	case ENTITY_COMMAND_MOVE:
		EntityMoveMessage move_msg = (EntityMoveMessage)msg;
		//treat values as components of force vector
		this.setAcceleration(move_msg.getX(), move_msg.getY(),
		move_msg.getRotation());
		break;
	default:
		break;
	}
}
```

And that's pretty much how the message system works. Now to the actual game engine.

Game Engine Layout

This is roughly what types of objects interact inside the engine. Note that each rectangle corresponds exactly to a specific class, except for “Hooks.”



The engine has a Model-View-Controller architecture, which basically means that the objects that contain data (Level, Entity), the objects that manipulate data (Controller) and the objects that draw the data (View) are separate from each other. This isn’t really important, but if you guys have stumbled upon the term in a software engineering class then it might better explain how the engine works.

State Machine

This is the highest level component of the game engine. It really doesn't do anything except that it lets the game switch from one mode (i.e., state) to another. For example, when the game is first initialized, you are at the menu (Menu State); when you click the "play" button, you then transition to the actual game (Game State).

For simplicity's sake I didn't draw out the inner workings of the End State and the Menu State, as they are pretty simple and they're at the bottom of the priority list in terms of development.

Game State

This is pretty self-explanatory. This class is only useful because it contains the Level object.

Level

This is probably the most important class in the engine. As the name implies, this class contains everything needed for exactly one level in the game. Among its components include: a list of all Entities in the level, a list of all Entity Controllers, map data, a resource database, and a particle engine. Among other functions, the Level object calls the other components of the engine to update once every frame, updates the physics of Entities, and checks/resolves collisions between Entities.

Map

This is pretty self-explanatory; this class contains data about level layout.

Entity

Entities are basically the objects within the level. Entities can include enemies, players, pick-ups, and bullets. Entities do not really do anything on their own, except keep track of their own data.

Model

Models are the graphical representations of an Entity. They include data such as what shape an entity is and an image of an entity.

EntityType

EntityTypes, as their name implies, is the "template" for Entity objects. They include data such as the base model of an entity, physics attributes, and entity-specific variables (ex. the EntityType subclass of the Player entity has a variable called "max_health"). The Level object uses the EntityType as a template to create new Entities.

Controller

Controllers are the objects that manipulate entities. Controllers that have the "hivemind" property control many objects at once, which is useful for manipulating entities with simple functionality.

Sounds, Sprites/Images, Tilesets

These are various resources used throughout the game. Sounds and Sprites/Images are self-explanatory; Tilesets are used to draw the map of the level.

Resource Database

This object stores all of the resources used the game. Among the various resources is stores include Models, EntityTypes, Sounds, Sprites/Images, and Tilesets. Having a resource database creates a standard way of accessing resources, which makes it easy for other components of the engine to access resources; for example, when a player gets hit, its PlayerController might query the database for a specific sound file and then play that sound.

Particle Engine

A Particle Engine basically lets the engine perform fancy visual effects. For example, when firing a bullet, the particle engine would draw a trail.

View

The View object functions like a "camera" pointing towards the Level object. It draws the map of the level (using a Tileset), draws the objects in the map, and draws the particles in the Particle Engine.

Hook

Hooks are basically any objects that listen for events during the game and perform some kind of special function. For example, the GUI would be a hook that listens to the event when the player gets hit by a bullet, for example.

COMPONENTS NOT INCLUDED – I didn't include these components because they're somewhat ancillary to the actual game engine. We still have to code these, though, however.

- Input system (i.e., Keyboard/mouse) – this will function as a special type of entity Controller.
- Networking system – this will also function as a special type of entity Controller

COMPONENTS NOT YET IMPLEMENTED – These are components I've described above but are still not implemented. They need to be written soon!

- Map and Tileset
- Resource database
- Particle Engine
- State Machine

That's pretty much it. Hopefully this makes the code a lot clearer and more understandable. Send me a message if you have any questions about anything I've talked about. Thanks!