BuildEngine Manual

This is a rough first draft of the documentation on BuildEngine. All items are at liberty to change.

This manual contains the basic information on the BuildEngine-GL project (current up-to-date version at the top of the page).

1. Quick Start

1.1 Setting up BuildEngine

To start writing code and using the engine, we first need to set up java so we can access the build engine classes and functions. First, download the latest stable version of build engine on buildengine.net. Then, depending on your IDE, add the jar files to the dependencies of your project. If you are using Gradle add the code below.

```
repositories {
    flatDir {
        dirs 'lib'
    }
}

dependencies {
    implementation name: 'BuildEngine v1.1'
}
```

Listing 1.1

Now that you've added BuildEngine to your project, you can test if BuildEngine is active by creating a main function, and printing out the welcome message of BuildEngine (see Listing 1.2).

```
System.out.println(BuildEngine.welcome());
```

Listing 1.2

If the current version of BuildEngine is printed to the console, you know build engine is working correctly.

1.2 Setting up a scene

To set up a scene, create a new class that extends the Scene object. Overwrite the begin() method and add Actors and Directors to the scene in this method. This can't be done in the main method, for there is no OpenGL

context in the main thread. The scene itself has no further functionality. To add functionality to the scene, use a custom Director, and add it to the scene using the begin method.

```
public static void main(String[] args) {
    BuildEngine.create();

    MyScene scene = new MyScene();

    BuildEngine.getEngine().getStage().setScene(scene);
}
```

Listing 1.3

1.2.1 Adding actors

To add actors to a scene, use a custom Director and overwrite the begin(); method (or overwrite the begin method in the scene), and add actors using the add(Actor actor); method. Actors can not be added to the scene directly from the main method, because the SpriteMask component now requires an OpenGL context, so it can generate the vertices. Example actor constructor:

```
new Actor("Actor 1", new Transform(new Vector2f(2,2), 10, 11, 0),
          new RigidBody(false));
```

Listing 1.4

The name of the actor is only used for debugging purposes, and doesn’t have to be unique. The transform is added using the default Transform constructor. Notice the 0 at the end, which indicates this actor will be located on the default zIndex.

1.2.2 Sprites and Rendering

By default, Sprites are rendered by the BatchRenderer director (added automatically to a new scene). Sprites hold a colour, a texture and uv-coordinates (full texture by default). The colour is only used if the texture is null. The UV-coordinates are given the whole texture by default. The ResourceManager class is used to load textures and spritesheets. SpriteSheet objects hold multiple sprites, cropped from the same texture using the UV-coordinates. A smart constructor can automatically divide the texture and extract the sprites. Make sure however you call

```
ResourceManager.createSpriteSheet(String key, SpriteSheet sheet);
```

in a loading method to create sprite sheets. After that you can recall the sprite sheet by using the key given and getting a sprite by index.

Sprites are given a centre-anchor point by default (0.5, 0.5). The anchor point describes how the sprite is position relative to the transform position of the owner. Each sprite's anchor point can be customised by using the setter in the class.

1.2.3 Collisions

To use the collision system use the BoxCollider component. The collision-registerer director will look for any BoxCollider component, register their collisions inside their array list of Collisions and execute all CollisionListeners added to the BoxColliders.

1.2.4 Input

The static class Input is used for registering keyboard and mouse inputs. Methods in these classes are self explanatory.

2. Documentation

This chapter documents the different systems of BuildEngine and how they work. We go in a lot of depth, but also give some practical examples on how to implement and use the features of buildengine.

2.1 Architecture

BuildEngine uses an implementation of the increasingly popular ECS system. However, because BuildEngine is written in Java, and some optimization features of ECS don't apply to Java, the engine functions a bit differently from normal ECS systems. You can think of BuildEngine's naming convention as if the player were attending a theatre show. In this theatre show, there is a Stage. The theatre show (the game) is being presented on this stage, divided into scenes. Each scene has actors who play in the scene, and directors directing the scene; telling the actors what to do. So if you ever get confused by a name in BuildEngine, think back to this example. Now, laid out below is a diagram of the engine's core structure (Figure 2.1).

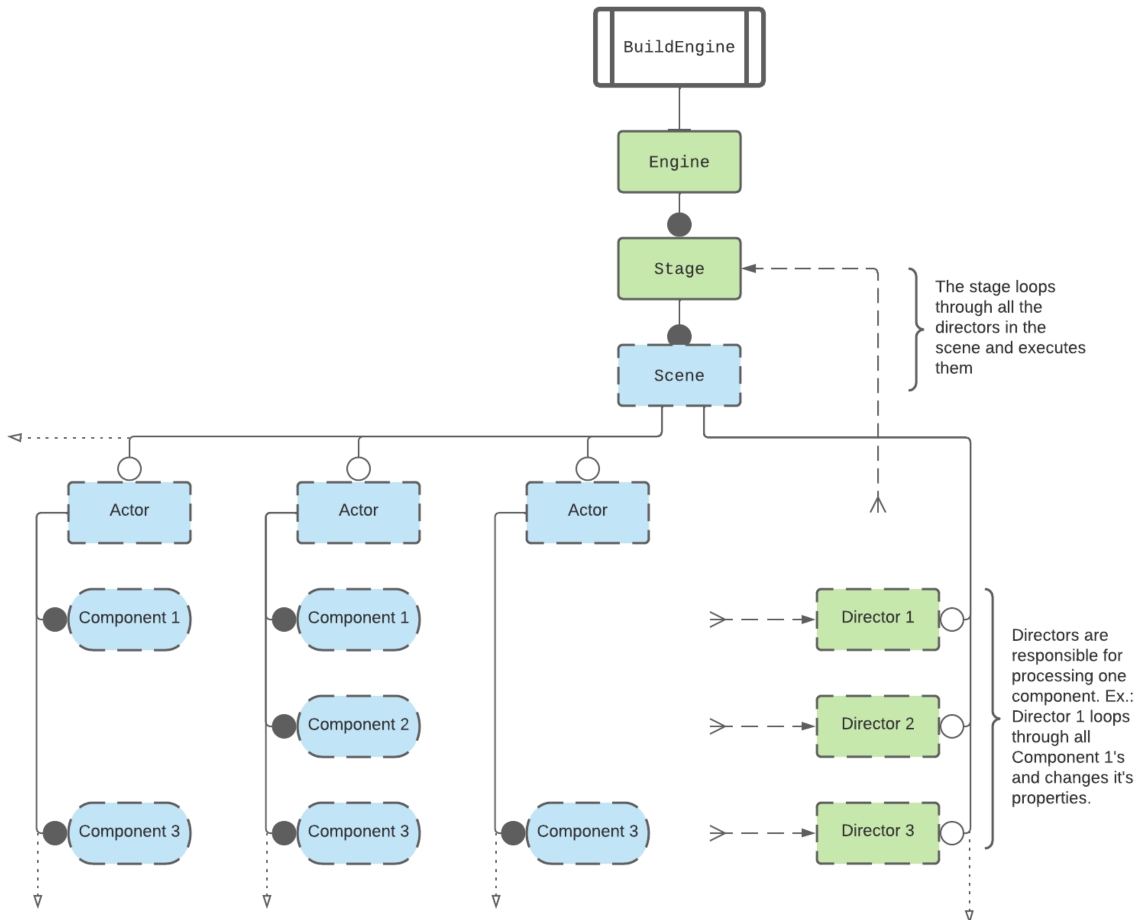

Figure 2.1

Before we continue, we have to take a look at the way Figure 2.1 is constructed. This is a layout of the core engine. Each box represents a class/object in the engine structure. When a box has a green colour, the class is an executing class, whereas if a box is blue, the class is a containing class. Containing classes only contain information. The class can't actually change its own data (except through functions contained in the class) and relies on executing classes to change its data. Executing classes have functionality in the engine. For instance, all executing classes can access the update() function. Some boxes have solid borders; these classes are created by the Engine, and the user (most of the time) doesn't overwrite them. Most boxes however don't have solid borders. These classes have to be created by the user. The boxes are connected with lines. If the line ends in a circle, the parent class contains an instance of the class the circle touches. White (open) circles means it doesn't matter how many instances of the class the parent class contains. A black (closed) circle means the parent class can only contain one instance of the class. The three small lines combining into one line are input lines, retrieving multiple classes and processing it (like with the director class).

The Engine class contains the main game loop. It creates the window, the game thread and initialises OpenGL on that thread. Finally it creates a new Stage object.
This new Stage object is, like the Engine, an executing class. It has all the executing functions (begin(), update(), fixedUpdate(), render() and cleanUp()). This class, unlike the `Engine` class, can be overwritten by using the `swapStage()` function in the Engine class. Earlier versions of BuildEngine used to work this way. The `Stage` (back then State) class contains all the essential executing functions to build a game architecture on your own. If you need really low level control, you can use the Stage class.

Everything bellow the Stage class, we still need to implement ourselfs, but we don't need to create it. The next class in line is the scene class. If you used other game engines before, this name shouldn't trip you up. The Stage class can hold one Scene. You set the initial scene using the queueScene() function in the Stage class. This triggers a transition and switches scenes.

The Scene class is the first containing class. This class exclusively holds information, and has no executing functions. It has essentially two lists of items. The first one is a list of Actors. All objects in the game are Actors (a.k.a. Entities or Objects in other engines). Actors in turn hold a list of Components, describing the behaviour and capabilities of the Actor. These are, just like the Actor itself, just containing classes. Actors and their components do not have any functionality themselves. That's where the second list of a Scene comes in: Directors. Directors, as shown by the green colour in Figure 2.1, are executing classes. These classes have access to all executing functions and, importantly, have access to the scene object they are added to. Because of this, they have access to all actors and their components.

Directors, therefore, have the responsibility of creating, changing and rendering all the data of the current scene. In principle, according to the theory behind ECS, one director should only be responsible for updating the one corresponding component. For example, if the game has a MoveComponent, the MoveDirector should be the only one changing the values inside the components. In practice however, this isn't always ideal, and since java isn't even able to take advantage of the efficiency boost this practice would provide, it isn't a rule set in stone. Now that we are all the way at the bottom of Figure 2.1, let's loop back around to the stage. As noted in the diagram, all the directors are executed on the stage. The diagram can however be misleading in this regard. The way the Stage accesses the Directors is of course through the List in the current Scene object, in the same way Directors access Actor's components through the Scene object. But in the diagram it seems like they magically collect them. And keep in mind, if the user has overwritten the default Stage object, all the lines from and to the Stage object become obsolete.

Now that we have a general idea of how the engine functions, let's take a look at a few examples of implementations of this structure.

```
 public static void main(String[] args) {
   BuildEngine.create();

   Scene scene = new Scene("GameScene");
   scene.addDirector(new BatchRenderer(), new Testing());

   BuildEngine.getEngine().getStage().setScene(scene);
}
```

Listing 2.1

In Listing 2.1 we can see a very simple implementation. First we create the Engine and stage class (by means of the first function). Then we create a new scene object (remember Figure 2.1; the border around the scene class is not solid, so we need to create them ourselves). After that we add two directors: a BatchRenderer director, who is responsible for rendering SpriteMask components, and our own custom Testing director. Remember: we can't add Actors yet, because of the OpenGL context. Actors always need to be created inside a Director. After that we navigate to the current stage (if you find navigating this function confusing, refer to Figure 2.1; we go from the static BuildEngine class, down the diagram, to the stage object) and set the scene to the scene we just created.
This is all we need to do to make the engine render a scene. But we don't have anything in our scene for our director to manipulate or render, so let's take a look at the code in our Testing director.

```java
private int index = 0;

@Override
public void begin() {
	ResourceManager.createSpriteSheet("player", new SpriteSheet(
		ResourceManager.getTexture("assets/images/player.png"),
		new Vector2i(64, 64), 13, null));

	SpriteSheet sheet = ResourceManager.getSpriteSheet("player");

	Actor player = new Actor("Player",
		new Transform(new Vector2f(), 5, 5),
		new SpriteMask(sheet.getSprite(0)));
	player.setDynamic(true);

	scene.addActor(player);
	scene.addActor(new Actor("Brick",
		 new Transform(new Vector2f(6, 0), 3, 3, -1),
		 new SpriteMask(new Sprite(
		ResourceManager.getTexture("assets/images/brick.png")))));

	Scheduler.addEvent(new RepeatingEvent(() -> {
	    if(index++ >= 8) {
		index = 0;
	    }
	    player.getComponent(SpriteMask.class)
				.setSprite(sheet.getSprite(index));
	}, 80));
}

@Override
public void update(float dt) {
	player.getTransform().getPosition().add(dt * 2.5f, 0);
}
```

Listing 2.2

This Listing 2 is a class called Testing, which implements MonoBehaviour. This interface contains the standard executing functions: begin(), update(), fixedUpdate() and cleanUp(). Those last two don't require implementation. First we begin by making a variable index to animate our player moving across the screen. Then we enter the begin method. Here we create a spritesheet using the useful ResourceManager class. (Don't worry if you don't understand how these specific functions work. Later in this documentation they are explained in depth). After storing the created sprite sheet in a variable, we create an actor called "Player". We're giving him the transform location of 0,0 (by leaving the Vector2f constructor empty), making his size 5x5 units and adding a SpriteMask component, passing the first image of the sprite sheet as default sprite. After that, we set the player to dynamic, so that their position and sprite changes are captured by the GPU. After that we add the player and another Actor to the scene. At the end we will add a new repeating event, which increases the index variable and updates the `SpriteMask` of the player. This way we will have a little animation. Finally we have an update function moving the player to the right.

This code makes use of a lot of more complicated features, but is a nice introduction to the way BuildEngine thinks, and how you should add actors and components.

2.2 Directors

There are some default directors added to the scene (see the begin() method in Scene.java). To create a custom director, create a class that extends the Director class. To add functionality implement either just Loader or MonoBehaviour. Loader only gives you the ability to use the begin and cleanup method. MonoBehaviour gives you access to the loader interface, and update(), fixedUpdate(). Update will be called as much as possible. Fixed updates will be called 60 times per second. This is also the function that the default Physics system uses.

2.2.1 Execution Priority

Since build 9.2021.2 execution priority can easily be changed inside the constructor. Execution priority tells the engine when to execute this director in relation to others. As of now the engine has three phases to execute: PRE, DEFAULT and POST. The PRE phase is for all directors that need to be updated first, for example input or collision registering. The DEFAULT phase is for all the normal functionality. The POST phase is for processes that need to be updated at the end, like resolving collisions. Be aware that this is another way to separate activity, next to the existing way of having multiple methods. The begin() method will always be called before the update() method, but the begin() methods of directors with execution priority POST will be called after the begin() methods of directors with the execution priority DEFAULT, but before all update() methods, regardless of the priority of the director. Se bellow for example

```
    begin() {
        begin PRE directors
        begin DEFAULT directors
        begin POST directors
    }

    update() {
        update PRE directors
        update DEFAULT directors
        update POST directors
    }

// Note that fixed update will not always be called after update(), for // fixed update will only be called 60 times per second, and update()
// much more often.
    fixedUpdate() {
        fixedUpdate PRE directors
        fixedUpdate DEFAULT directors
        fixedUpdate POST directors
    }

    render() {
        render PRE directors
        render DEFAULT directors
        render POST directors
    }
```

Figure 2.2: Executing order

2.3 Rendering

The default render engine uses an implementation of OpenGL Vertex rendering. It uses a default shader (as of v0.1 not yet overwritable) and a Batch rendering system. The `BatchRenderer` is automatically added to every newly created scene (provided the `Scene.begin()` method is not overwritten). This class contains a list of `RenderBatch` classes. It makes use of texture streaming, and creates new batches if there are no other batches available (when: No batches exist, existing batches are full or have too many texture channels occupied).