Setting up the developing environment

See Setting Up the Developing Environment

Building

Bullet-Inferno uses the default build process of Eclipse, please see Setting Up the Developing Environment for how to install Eclipse.

Coding Guidelines

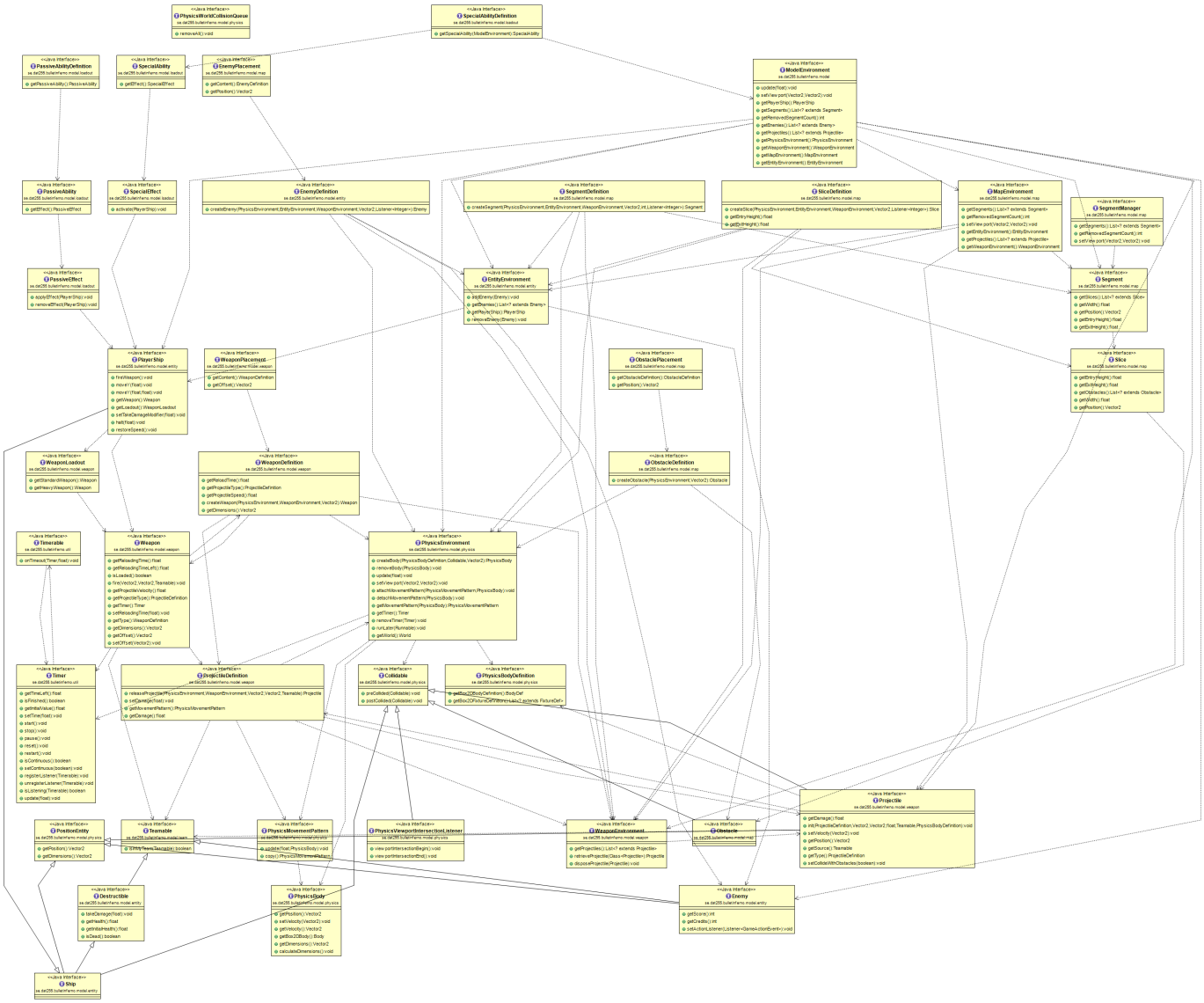
Please see our Coding Guidelines for how we expect code to be written for Bullet-Inferno.

Application components

The application uses a passive MVC structure (meaning models do not send events to views; instead views continuously poll models for data), which consists of three major components: **models**, **views** and **controllers**. This also reflects how the packages are structured throughout the application.

UML

A raw outline of the structure of Bullet-Inferno as UML can be seen below.



Models

The models in Bullet-Inferno are representations of game objects, i.e. things that actually appear in the game in one way or another. Models must not know about neither views nor controllers.

Environment

The models are broken up into multiple sub-systems, that we called environments. Each environment holds the models associated with it and provides a single entry point to that collection of models. The environments are themselves held by a sort of master environment called `ModelEnvironment`.

Views

The views in Bullet-Inferno contain the drawing specifications for models that actually can be seen in the game. They are to poll their respective model(s) for any data needed to represent said model(s) on the screen. An important property that the views must consider is the model's hitbox, as the view must make sure to try to match the hitbox. The views are allowed to know about their models and other objects necessary to render their screen (such as the current viewport), but they

should not modify the state of the application.

Controllers

The controllers in Bullet-Inferno take on the role of both updating models and telling views to render themselves. They also handle user input. Controllers know about both models and controllers.

The current controller hierarchy starts with a MasterController that essentially only forwards render calls to the currently active **screen**. The MasterController also takes care of handling the android resume/pause events (see Managing the Activity Lifecycle).

Screens

The application is further separated into multiple screens as to try and reduce the complexity of the project. A screen is a type of controller that represents a specific application screen, such as the loading screen or the game screen. An important thing to note about the screens is that they do not share the same camera. This is to, for example, allow the menu screen to place its elements at absolute pixel coordinates while the game screen uses an abstracted metrical size for its representation.

Definitions

Throughout the projects we have multiple so called `Definition`s. These are essentially predefined factories for the object(s) in question. We have chosen to use Enums as the main implementation of the definitions, as it turned out to be the convenient way for us.

Miscellaneous information

Game size and placements

The game uses a virtual screen size of 16m x 9m for the models. It is up to the Game Screen to translate between these coordinates and the real size of the device.

Texture handling

Throughout the project (loading of) textures are handled in two different ways. The recommended, and almost exclusively used, is by using the `ResourceManager` implementation. The second alternative is manually handling everything.

ResourceManager(Impl)

The `ResourceManagerImpl` (referred to as the resource manager) class wraps the `AssetManager` from libGDX. The AssetManager provides, among other things, a way to asynchronously load textures. It also makes it possible to dispose all loaded textures more easily. The resource manager in turn further extends this functionality.

The main features of the resource manager is that it maps identifiers to textures, meaning our

models does not have to keep track of what texture they are, but only their identifier. This is to keep the models separated from the views. To add a texture to the resource manager, the only thing one has to do is adding the identifier and the corresponding texture path to the `TextureDefinitionImpl` enum's mapping.

Another important feature that the resource manager provides is that it allows for multiple resolutions (**480x270**, 800x450, 1280x720 and 1920x1080, where 480x270 is the default) for a single texture path. It also automatically loads the best match depending on the current device. This requires no additional changes to the code.

To set up a texture with multiple resolutions, what one has to do is add the default (480x270) resolution texture to the appropriate folder of the `Bullet-Inferno-android/assets` folder, and then adding the texture for the other three other resolution to a subfolder named after the resolution in question (**/800450** for 800x450, **/1280720** for 1280x720, etc). **Note** that the default resolution texture is placed in the base directory and not in a subdirectory.

Manual handling

Manually handling textures is not advised. In the cases where it is necessary (like the `LoadingScreenView`, as it is created before the resource manager is loaded) **one must remember to manually dispose of all textures loaded this way**.

Design Decisions

Android API level

Bullet-Inferno is built to target Android API level 17, and set to allow Android API level 8 as the minimum.

The reasoning behind having level 8 as our lowest allowed version is that Bullet-Inferno uses OpenGL ES 2.0 features, and the lowest version which supports that is API level 8. OpenGL ES 2.0 enables you to use some cool effects, but the most significant advantage is that texture dimensions don’t have to be a power of two (e.g. 64x32, 16x128, etc). That allowed us to spend less time on creating graphics for the application, as resizing everything to a power of two would indeed have resulted in a lot of time wasted. Furthermore, not even the new Google Play Store app supports versions below level 8.

The reason we choose to target API level 17 instead of 18 (the newest version at the time of developing this application) is that level 18 ships with OpenGL ES 3.0, a version we have no means of testing and therefor can not guarantee that our application will run as expected.

JDK (java) version

Bullet-Inferno is set to use JDK version 1.6. This is because we have no use for any of the 1.7 features, and while 1.7 can still compile 1.6 code the other way around does not work.

External dependencies

libGDX (see the libGDX wiki for documentation)

The application uses libGDX because mainly because it allows for cross-platform deploying. This is incredibly useful for testing as it does not require the developer to have an Android device. The platform-specific details are kept to a minimum, practically only coming down to handling touch, considering memory handling/limitations and pausing/resuming the application.

Furthermore, libGDX takes care of the low-level OpenGL details in a smooth way. Displaying an image is as easy as loading a Texture from a directory, putting it in a Sprite and then calling sprite.draw(x, y) (in reality the application uses a SpriteBatch to make drawing more efficient, but it's the same idea).

Moreover, libGDX comes with Box2d built-in, which the application uses to take care of collision detection. An adaptation of Box2d was implemented to suit the application's needs better. The use of Box2d has one critical implication: it makes update methods for models redundant, as Box2d already takes care of that. Therefore, each object that needs a hitbox is equipped with a Box2d Body which has a velocity. That means that game objects' velocity can only be manipulated through the use of forces, which Box2d facilitates.