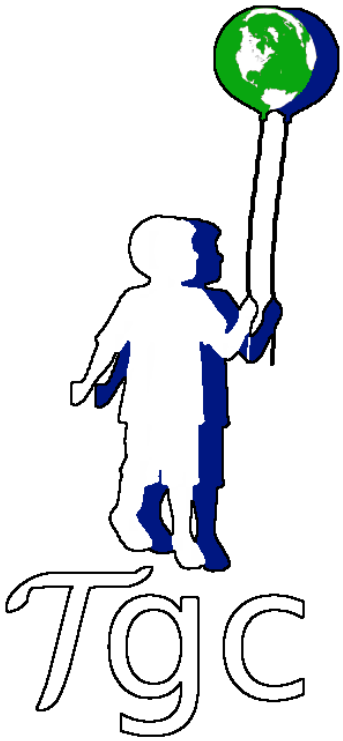


Heart & Hand
Center for Youth and Their Families

Denver Defenders

Client: The Giving Child nonprofit
Heart & Hand nonprofit

Team Members:
Corey Tokunaga-Reichert, Jack Nelson, Kevin Day, Milton Tzimourakas, Nathaniel Jacobi

Introduction

Client Description:

The Giving-Child is a group operating in Denver that wants to inspire fortunate children to help out other kids their age who come from disadvantaged backgrounds. They aim to empower kids to help change global issues by making a positive, immediate impact. To achieve this, the Giving-Child believes that the easiest way to connect with a kid is through games because of how well they maintain a child's attention, and how engaging they can be to children.

Heart & Hand is a charity group operating in Denver which runs a center for at risk youth and their families. They are committed to helping disadvantaged youth with educational support. They offer a variety of programs in Denver to help children get the education they need as well as provide safe environments for the children.

The Giving Child approached Colorado School of Mines to have some students create a game engine which could be used to create a variety of short, 2D mini games with relative ease. They would then utilize this engine to create a large variety of fun challenging mini games for kids to inform them and help inspire them to make a difference, while giving some proceeds to the Heart and Hand organization.

Product Vision:

The purpose of the game engine we created is to target children and to raise awareness amongst them about some of the problems that other less fortunate kids in their community are experiencing, such as needing food or shelter. By having the player assume the role of a superhero that is actively exploring a city and helping others that they find along the way, kids playing the game will begin to make connections about how they can help others that they see in real life. The mini games that players trigger inside the maze for helping a child are also created and designed with this purpose in mind and require players to accomplish some sort of helpful task in a mini game such as preparing a salad or putting out a fire.

Requirements

Functional:

- Game Engine:
    - Load Levels
    - Update game objects properly
    - Evaluates win/lose conditions
    - Handle user inputs and actions
    - Navigate between screens
    - Easily create new attributes and win/lose conditions
- XML reader
    - Parses XML file to create a game screen from objects, attributes, win conditions, and lose conditions
    - Sandbox style game object to dynamically create objects with only specified attributes
- XML writer
    - Should export mini-games as XML documents
    - Format should follow the same format as the reader
- Maze
    - Allow the user to move around
    - Have the children spawn in randomly in the proper locations
    - Children follow after winning a game

    - Children move to a random trigger point after losing a game
    - Collecting all of the children wins the maze
    - Losing too many times ends the maze with a loss

Non-Functional:

- Should be multi-platform for mobile
- Should be built using LibGDX and Gradle
- Should have a nice User Interface for mini-game creation
- Server to upload/download mini-games to and from
- Tap/swipe/pinch/touch controls
- Show facts given by the client between screens
- The games
    - Should spread the message of Giving Child and Heart and Hand
    - Be built using a level editor from the engine
    - Quick, simple, and engaging

XML Integration:

- User defined tags for objects
- Manipulate game engine values (game speed, difficulty)
- Define games using XML to be loaded by “main” game

Risks:

- Failing to deliver The Giving Child Message through mini-games
- Game Design
- Building a flexible engine that can build a variety of mini-games
- Outputting games in a standardized XML format
- Getting iOS build support to work properly
- Unfamiliarity with LibGDX/Gradle

System Architecture/Technical Design

Various components:

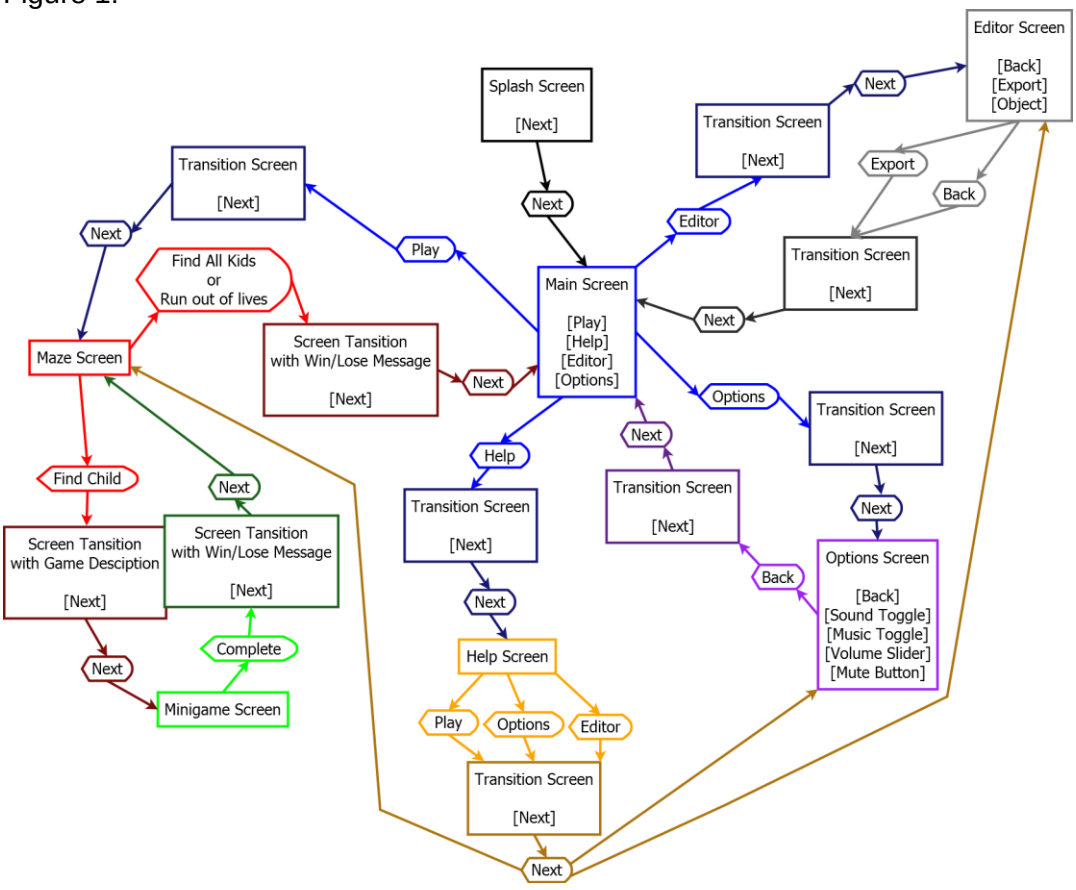
- XML writer/reader
    - The XML reader is responsible for parsing through the .xml files which store all information on each individual mini game and converting them into Level objects. The reader dynamically creates each game object based on the specifications within the .xml files.
    - The XML writer is responsible for converting Level objects back into .xml files for easy and accessible storage of mini games.
- Level Objects
    - Level objects are at the core of the game, they store all pertinent information for each level. Most of this information is in the form of GameObjects, Win conditions, and Lose conditions. Each level has a unique combination of these to create a specific mini game.
    - The Win and Lose conditions can be easily modified to allow for unique game types.

- Game Objects
    - Each Game Object contains a list of attributes and listeners which allow it to behave in the desired fashion and react to other objects and input from the user.
    - The type of attributes and listeners can be easily modified; this allows new functionality to be implemented with relative ease.
- Level Editor
    - Creates a grid system to hold the objects
    - Prompts the user for a level name
    - When you spawn an object the attributes and listeners can be selected.
    - When an object is placed it goes to the close grid location.
    - Once the user is finished they can press the export button to generate an XML file for the level.
    - If the eclipse project is refreshed, new levels will be loaded and can be played in the maze.
- Maze / Maze screen
    - Serve as one of the user's main play screens
    - Children in the maze are entry point to mini games
    - Players will navigate around the maze and complete it after saving all the kids

Screen Flow:

Figure 1 below shows the paths between the screens and what actions it takes to get to them. The words in brackets are the buttons that can be pressed. Each screen has its own color and the transition screen that follows is a darker shade of that color, to better illustrate the path taken.

Figure 1:



UML (first design):

Figures 2-4 are the initial UML's that were made after two weeks of working on the project. Most parts remained until the end, but some were heavily changed or were eliminated entirely.

Figure 2

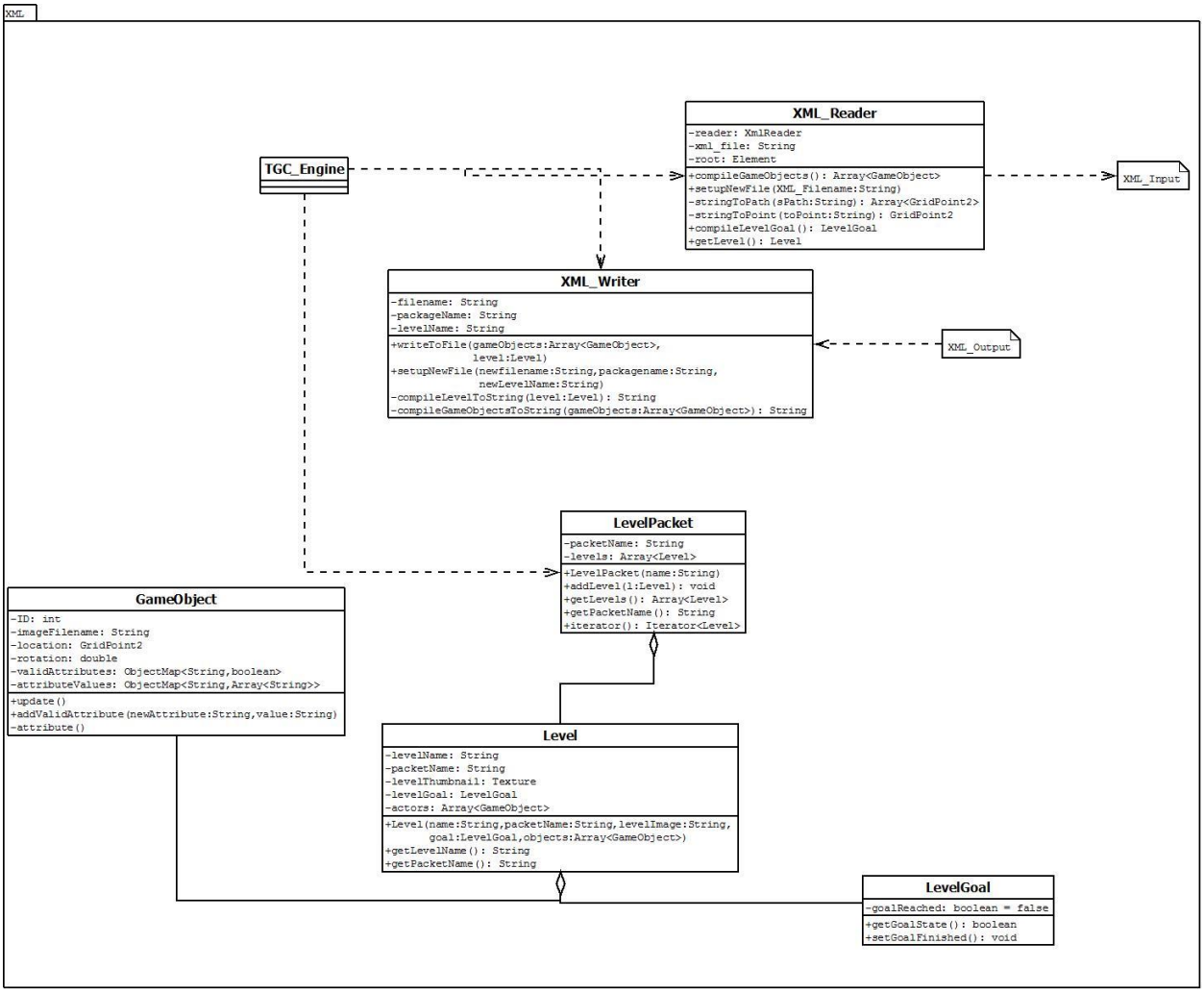


Figure 3

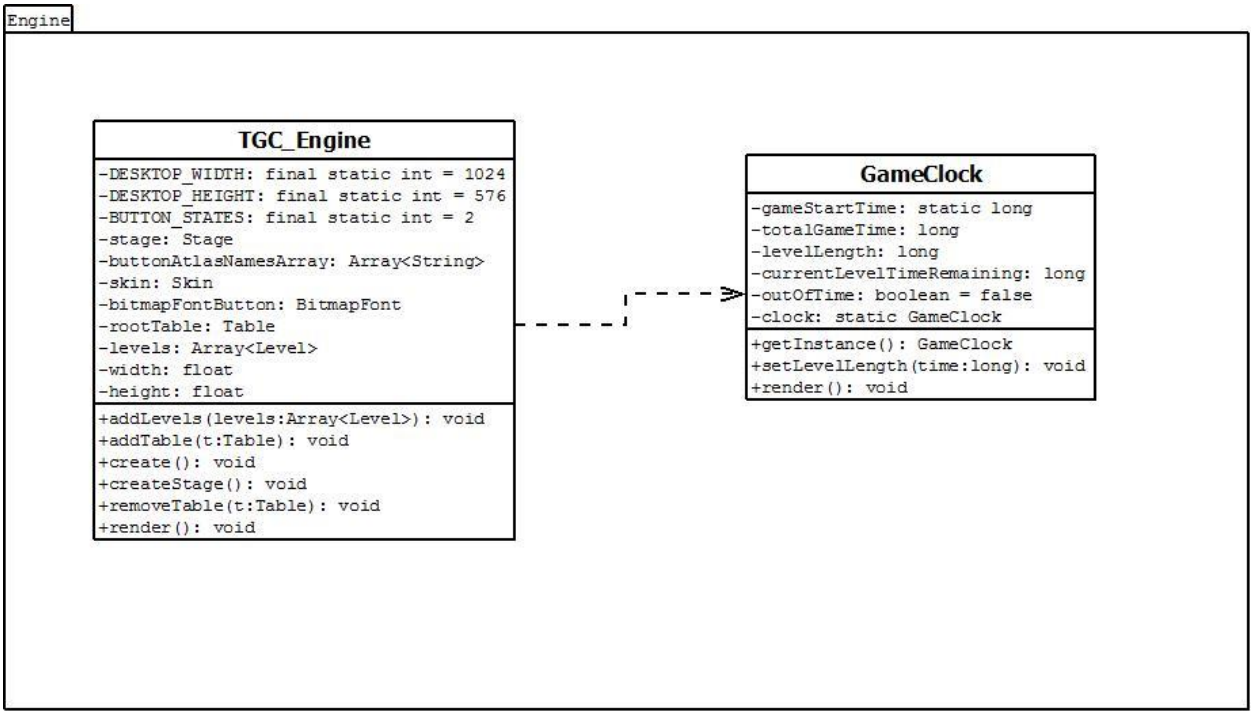
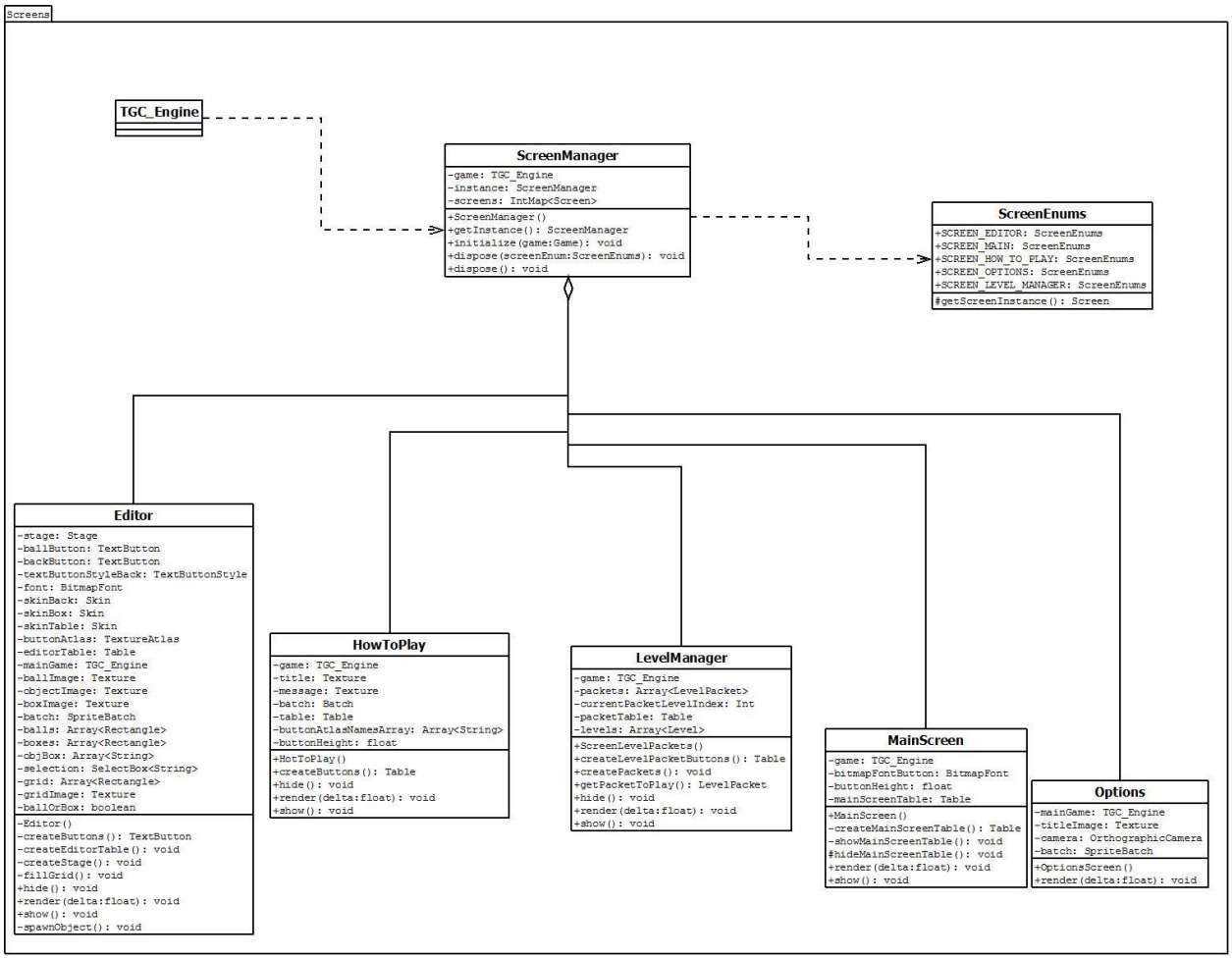


Figure 4



UML (Final Design):

Figures 5-7 are the final UML's. Figure 7, which is of our Screens package, is a skeleton UML since the actual UML took up too much space to be readable in the document, but it's format is very similar to the first UML in Figure 4.

Figure 5

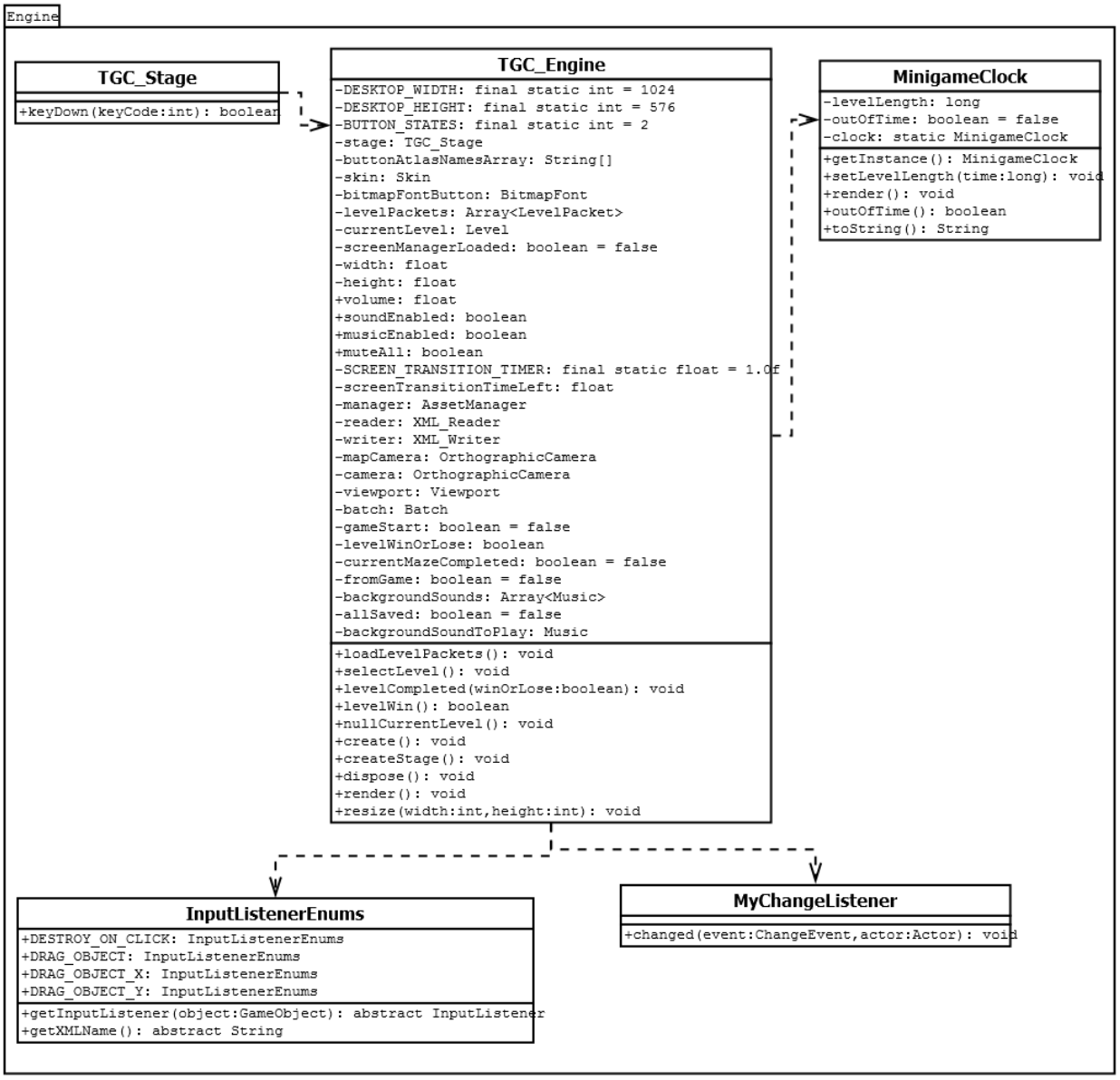


Figure 6

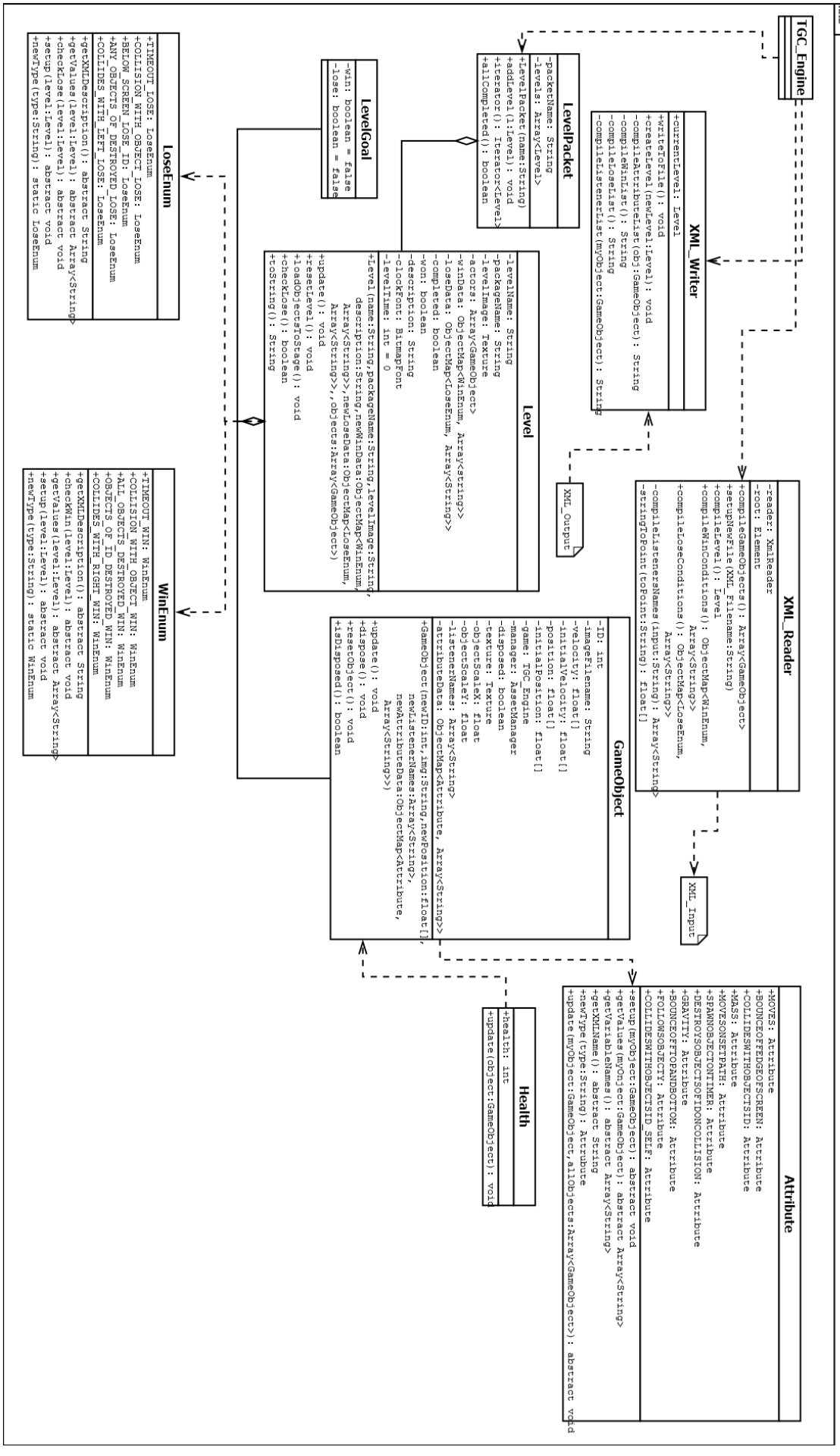
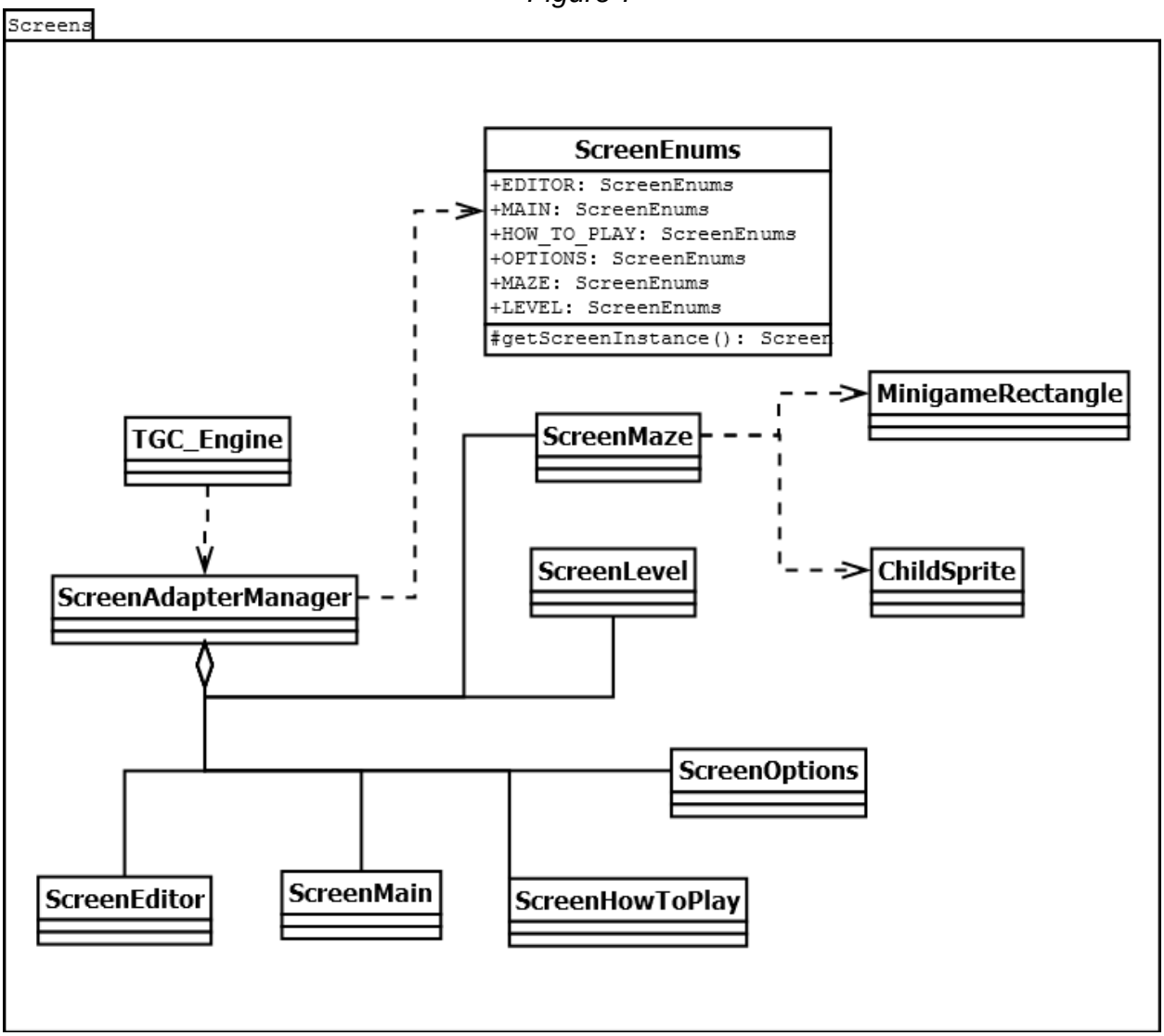

XML
TGC_Engine
XML_Reader
-reader: XmlReader
-root: Element
+compileGameObjects(): Array<GameObject>
+setupNewFile(XML_Filename:String)
+compileLevel(): Level
+compileWinConditions(): ObjectMap<WinEnum, Array<String>>
+compileLoseConditions(): ObjectMap<LoseEnum, Array<String>>
-compileListenerNames(input:String): Array<String>
-stringToPoint(toPoint:String): float[]
XML_Output
XML_Input
XML_Writer
-currentLevel: Level
+writeToFile(): void
+createLevel(newLevel:Level): void
-compileAttributeList(obj:GameObject): String
-compileWinList(): String
-compileLoseList(): String
-compileListenerList(myObject:GameObject): String
LevelPacket
-packetName: String
-levels: Array<Level>
+levelPacket(name:String)
+addLevel(l:Level): void
+iterator(): Iterator<Level>
+allCompleted(): boolean
LevelGoal
-win: boolean = false
-lose: boolean = false
Level
-levelName: String
-packageName: String
-levelImage: Texture
-actors: Array<GameObject>
-winData: ObjectMap<WinEnum, Array<string>>
-loseData: ObjectMap<LoseEnum, Array<String>>
-completed: boolean
-won: boolean
-description: String
-clockFont: BitmapFont
-levelTime: int = 0
+Level(name:String,packageName:String,levelImage:String, description:String,newWinData:ObjectMap<WinEnum, Array<String>>,newLoseData:ObjectMap<LoseEnum, Array<String>>,objects:Array<GameObject>)
+update(): void
+resetLevel(): void
+loadObjectsToStage(): void
+checkLose(): boolean
+toString(): String
GameObject
-ID: int
-imageFilename: String
-velocity: float[]
-initialVelocity: float[]
-position: float[]
-initialPosition: float[]
-game: TGC_Engine
-manager: AssetManager
-disposed: boolean
-texture: Texture
-objectScaleX: float
-objectScaleY: float
-listenerNames: Array<String>
-attributeData: ObjectMap<Attribute, Array<String>>
+GameObject(newID:int,img:String,newPosition:float[], newListenerNames:Array<String>, newAttributeData:ObjectMap<Attribute, Array<String>>)
+update(): void
+dispose(): void
+resetObject(): void
+isDisposed(): boolean
Attribute
+MOVES: Attribute
+BOUNCEOFFEDGEOFSCREEN: Attribute
+COLLIDESWITHOBJECTSID: Attribute
+MASS: Attribute
+MOVESONSETPATH: Attribute
+SPAWNOBJECTONTIMER: Attribute
+DESTROYSOBJECTSOFIDONCOLLISION: Attribute
+GRAVITY: Attribute
+BOUNCEOFFTOPANDBOTTOM: Attribute
+FOLLOWSOBJECTY: Attribute
+COLLIDESWITHOBJECTSID_SELF: Attribute
+setup(myObject:GameObject): abstract void
+getValues(myObject:GameObject): abstract Array<String>
+getVariableNames(): abstract Array<String>
+getXMLName(): abstract String
+newType(type:String): Attribute
+update(myObject:GameObject,allObjects:Array<GameObject>): abstract void
Health
+health: int
+update(object:GameObject): void
LoseEnum
+TIMEOUT_LOSE: LoseEnum
+COLLISION_WITH_OBJECT_LOSE: LoseEnum
+BELOW_SCREEN_LOSE_ID: LoseEnum
+ANY_OBJECTS_OF_DESTROYED_LOSE: LoseEnum
+COLLIDES_WITH_LEFT_LOSE: LoseEnum
+getXMLDescription(): abstract String
+getValues(level:Level): abstract Array<String>
+checkLose(level:Level): abstract void
+setup(level:Level): abstract void
+newType(type:String): static LoseEnum
WinEnum
+TIMEOUT_WIN: WinEnum
+COLLISION_WITH_OBJECT_WIN: WinEnum
+ALL_OBJECTS_DESTROYED_WIN: WinEnum
+OBJECTS_OF_ID_DESTROYED_WIN: WinEnum
+COLLIDES_WITH_RIGHT_WIN: WinEnum
+getXMLDescription(): abstract String
+checkWin(level:Level): abstract void
+getValues(level:Level): abstract Array<String>
+setup(level:Level): abstract void
+newType(type:String): static WinEnum

Figure 7


Screens
ScreenEnums
+EDITOR: ScreenEnums
+MAIN: ScreenEnums
+HOW_TO_PLAY: ScreenEnums
+OPTIONS: ScreenEnums
+MAZE: ScreenEnums
+LEVEL: ScreenEnums
#getScreenInstance(): Screen
MinigameRectangle
TGC_Engine
ScreenMaze
ScreenLevel
ChildSprite
ScreenAdapterManager
ScreenOptions
ScreenEditor
ScreenMain
ScreenHowToPlay

Design and Implementation decisions

LibGDX:

LibGDX is the free graphics library we are using to create this game at the request of our mentor. He has familiarity with it and it allows cross platform development. The library allows a great deal of customization, which allows us the ability to implement whatever features we need to implement into our engine and bend it to our will.

XML Writer/Reader:

We used the .xml file format to store the data on all of our levels for a number of reasons. First because LibGDX natively supports it, second .xml format can be read and edited easily in any text editor, this allows us to debug any issues with the reader/writer with relative ease. Finally, the .xml file format is extremely flexible, this allows our team to change where and how the information is stored as the requirements for the mini games change as well as add completely new aspects to game objects, win conditions, and lose conditions.

The XML reader interprets the Level information and packs it into a Level object in order to be used by the GameEngine. The Reader allows for any number of files to be read in sequentially, this allows the GameEngine to read in all files and store them upon the game's startup.

The XML Writer unpacks the information stored in each Level created by the Editor and formats it into an .xml document. It receives information from the level editor sequentially and then writes them to their individual .xml files.

Game Objects:

Game Objects needed to be as flexible as possible to allow the future game designers to create as many different types of objects as possible. With this in mind, we had all of the object's information stored in various lists. First, they have a list of attributes. These attributes can describe any behavior of the objects, such as moving, colliding with the edge of the screen, colliding with other objects, destroying other objects upon collision with them, or any other functionality the user could think of. Next, objects have lists of "listeners" which allow the objects to respond to user input. For instance, we have a listener called 'drag_object', upon pressing the screen on the desired object's location; the object will then begin to track to the user. These easily alterable lists allow the future game designers to create a huge variety of game types.

Asset Manager:

The Asset Manager has built-in functionality through LibGDX that allows for asynchronous asset loading, while the main game runs on a separate thread. This allows for us to have smooth (depends on your interpretation) transitions between screens and levels while the needed resources for the next screen is being loaded. Without the Asset Manager, the transitions would switch from one event to another after loading all the assets on the main thread which would keep us from implementing a nice screen switching system.

Screen Management:

LibGDX has a ScreenAdapter class built in to the framework that allows for elements on a specific ScreenAdapter to be rendered to the main stage when a specific screens Show() method is called. While this allows ScreenAdapters to be switched between, the base functionality for switching between them was clunky, required editing some sort of array and directly addressing an index of an array to call it. This switching also allowed for multiple instances of a single ScreenAdapter to be accidentally created which causes issues with resource management and disposal methods.

In order to get around this clunky built in management, we created a ScreenAdapterManager class, a ScreenAdapterEnum class, and a ScreenAdapter class for each of the unique screens we needed layouts for. The ScreenAdapterEnum class was created to reference the individual ScreenAdapter classes and has an abstract method so that each enumerator returns an instance of the ScreenAdapter it is responsible for. The ScreenAdapterManager class has a map of the enumerators to the respective ScreenAdapters, allowing only one instance of each ScreenAdapter to be created within the manager.

The manager is accessed using a singleton strategy which makes sure there is only ever one instance of the ScreenAdapterManager allowing for easy resource management. When the program exits, the manager is disposed of, as well as disposing each of the screens that have been created. When a screen needs to be accessed, the show method of the manager is called by passing in a screens enumerator. The manager then hides the current screen, loads the screen transition effect, and displays the screen being accessed once the relevant resources have been loaded.

Level Editor:

The level editor will allow us to create levels more quickly than with writing individual XML files by "hand". It gives a visual aid for the position of the objects and you have direct control over the attributes it possesses for its final form. It pairs up with the XML Writer to write a proper XML file for the level that can very easily be loaded by the engine and be playable.

The editor itself is a screen managed by the ScreenAdapterManager class we implemented, so it has complete control over what goes on in itself. The buttons to access placing an object, going back to the main menu and exporting the level are hidden from the user initially. They are visible when the mouse is over a certain area and while the user is not placing an object. This allows more of the area to be uses and so there are no accidental presses. We only allow a user to place one object at a time so that they can apply the specific attributes they want to that object. Additional objects will be placed on a grid system (in the final product they will be less noticeable or hidden altogether) that way some snapping can be achieved so the user can have uniformity without the headache of trying to match pixels with something else.

Maze and MazeScreen:

The mazes were created as a tiled map using the Tiled Map Editor program, which was chosen because of its ease of integration with the LibGDX Library, along with being able to create a fully customizable map. Using the Tiled Program allows us to detect collisions between the maze and the players very easily, along with any other map or layer features that we need.

Results

Results Achieved:

- Editor:
  - Outputs XML files for levels
  - Can set varying attributes for different game objects.
  - Objects snap to a grid.
- GameObjects:
  - Can destroy on click.
  - Can be free dragged.
  - Can be drug on X axis only.
  - Can be drug on Y axis only.
  - Can collide with other objects based on ID.
  - Can bounce off borders.

- Game launches with company logos for The Giving Child, and Heart and Hand.
- Info dialogue screens between transitions.
- Lose conditions:
    - Time runs out.
    - Object with ID is moved to location.
- Maze:
    - Children trigger game packs.
    - Game packs return to maze after game is won or lost.
    - If game is won child follows you.
- ScreenAdapterManager to switch between active ScreenAdapters.
- ScreenAdapters to draw screens for:
    - Editor.
    - How to play.
    - Level.
    - Main.
    - Maze.
    - Options.
- Screen transitions to other screens with a curtain.
- User Interface that:
    - Is clickable.
    - Resizes.
    - Is skinned.
- Win conditions:
    - Destroy all objects.
    - Move objects with ID to location
    - Collision with Objects of ID
    - Timer Expires
- XML reader/writer:
    - Reads in the xml files to create game objects.

Constraints/Limitations/Details:

- Currently runs on Desktop, Android, and iOS
    - Can't actually test on iOS hardware (requires developer's license); iOS has only been tested on the emulator.
- Written in Java using Eclipse
- Uses the LibGDX library
    - Requires API 20 for Android as a minimum

Lessons learned:

- Merge conflicts are more prominent when working with larger teams.
- Learning new libraries in a short amount of time can lead to confusion, especially with limited documentation.
- Refactoring a large class can lead to broken code, due to forgetting to change how smaller sections of the class worked.
- Pair programming can help code be finished a lot faster, and with a better structure.
- When working on a project, there is a lot of extra stuff such as documentation, and meetings that end up reducing the amount of time that can be spent actually developing the code.

- Time management can be difficult when you are unfamiliar with your team's abilities, and areas that they excel or may fall behind in.

Appendices

Installation Instructions:

- Download and install Java 7 or higher.
- Download the Android SDK.
- Use the SDK Manager to download the tools and extras, as well as SDK 4.4.w.2 (API 20)
    - Remember the location you store the SDK, as the path will be needed later.
- Download and install Eclipse.
- Run eclipse, go to help -> new software. From here install the ADK tools from https://dl-ssl.google.com/android/eclipse/
- In Eclipse, go to help -> new software. Install the correct version of Gradle.
    - For Eclipse 4.4 or greater install from: http://dist.springsource.com/snapshot/TOOLS/gradle/nightly
    - For eclipse < 4.4 install from: http://dist.springsource.com/release/TOOLS/gradle
- Download and install Git Bash.
- Git clone the branch you want to work on.
- Create a file local.properties inside the TheGivingChildEngine/LibGDX_Generate/
- Add this line of text to the file: sdk.dir=F:/android-sdk_r24.2-windows/android-sdk-windows where sdk.dir= points to the path on the right of your Android SDK
- Use Gradle to import the folder LibGDX_Generate into your workspace, by following these steps:
    - Open Eclipse, right click in Package Explorer, left click "Import…"
    - The import source screen will open. From here select Gradle-> Gradle Project and left click "Next".
    - The import Gradle Project screen will open. From here left click "Browse…" and navigate to the LibGDX_Generate folder within the cloned git repository.
    - Click "Build Model", this generate the projects that can be imported.
    - Select all the generated projects under the LibGDX_Generate path, as well as LibGDX_Generate and lift click "Finish".
    - Wait for Gradle to import the projects.
    - You can now begin working on The Giving Child Engine through the core project.
- Changes made to the core project will be made to other projects as needed by Gradle.
- If refreshing the workspace does not work, select all project folders in Eclipse -> right click -> Gradle -> Refresh all.
- Changes to Assets should be done so through the Android project's Assets folder, which Gradle will push to the other projects.
- Do not push to the Master Branch if your code does not work. If you want to push code that is not ready to be merged, do so in a custom branch.
- If you are working on a separate branch and want to merge with Master, pull from Master, fix any conflicts, commit, then push to Master.

iOS Deployment (without a certificate):

- Install the RoboVM plugin for Eclipse and follow the instructions on the website.
- The installation guide will recommend increasing the heap space for the initial build and we recommend it, as it will prevent the app from running out of space.

- Once installed right click on the iOS project and run as an iOS application simulation and the iOS simulator should start up.

JavaDoc link:

The link to our JavaDoc of the entire project is here.

Or the URL is:
http://jackwesleynelson.github.io/TheGivingChildEngine/

Warnings:

Do not use indexed .png for assets, they will not load properly on Android, and likely won't load on iOS. The .png should be packed using the gdx-texture-packer which can be found in the git repository. If you receive errors when packing, enable rotation. If there are still issues, increase the maximum image dimensions.