

CONSTRUCTIVE PROCEDURAL CONTENT GENERATION
USING IMAGE DATABASES

by

Utku Gultopu

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
February 2019

© Copyright by Utku Gultopu, 2019

Table of Contents

| | |
|--|-----|
| List of Figures | iii |
| Abstract | v |
| Chapter 1 Introduction | 1 |
| Chapter 2 Background to the Angry Birds Game | 3 |
| Chapter 3 Image Capture and Pre-processing | 5 |
| Chapter 4 Algorithm | 13 |
| Chapter 5 Example of the Build Process | 33 |
| Chapter 6 Conclusion | 41 |
| Appendix A Level Screenshots | 43 |
| Appendix B Parameter Values | 64 |
| Bibliography | 65 |

List of Figures

| | | |
|------------|--|----|
| Figure 2.1 | A structure built using the PCG of Jiang et al. [4]. | 4 |
| Figure 2.2 | A structure built using the PCG of Ferreira et al. [2]. | 4 |
| Figure 3.1 | Flowchart demonstrating the process of vectorizing the raster image | 6 |
| Figure 3.2 | Input cow image | 7 |
| Figure 3.3 | Cow image converted to black-and-white | 8 |
| Figure 3.4 | Black-and-white cow image after being denoised | 9 |
| Figure 3.5 | Black-and-white cow image after being denoised and converted to Scalable Vector Graphics | 10 |
| Figure 3.6 | Vectorized cow image with a ‘polygon‘ element, instead of a ‘path‘ element | 11 |
| Figure 4.1 | White bird | 16 |
| Figure 4.2 | Blue bird | 17 |
| Figure 4.3 | Yellow bird | 17 |
| Figure 4.4 | Red bird | 18 |
| Figure 4.5 | Black bird | 19 |
| Figure 4.6 | Example of a structure with inadequate number of blocks for pig insertion | 24 |
| Figure 5.1 | The raw, unedited cow image | 33 |
| Figure 5.2 | Edited cow image | 34 |
| Figure 5.3 | Vectorized cow image | 35 |
| Figure 5.4 | The mesh on the cow image | 36 |
| Figure 5.5 | Individual loops in the mesh that will contain a primary block and that won’t | 37 |
| Figure 5.6 | Generated structure with only primary blocks | 38 |

| | | |
|-------------|--|----|
| Figure 5.7 | Generated structure with primary blocks and the inserted platform blocks | 39 |
| Figure 5.8 | Generated structure with primary blocks, the inserted platform blocks and pigs | 40 |
| Figure A.1 | Sample input image and output level: Cat | 44 |
| Figure A.2 | Sample input image and output level: Cow | 45 |
| Figure A.3 | Sample input image and output level: Dog | 46 |
| Figure A.4 | Sample input image and output level: Hedgehog | 47 |
| Figure A.5 | Sample input image and output level: Taj Mahal | 48 |
| Figure A.6 | Sample input image and output level: Triumphal Arch | 49 |
| Figure A.7 | Sample input image and output level: Parthenon | 50 |
| Figure A.8 | Sample input image and output level: Louvre Pyramid | 51 |
| Figure A.9 | Sample input image and output level: Grime | 52 |
| Figure A.10 | Sample input image and output level: Jack-o-Lantern | 53 |
| Figure A.11 | Sample input image and output level: Haunted House | 54 |
| Figure A.12 | Sample input image and output level: Candy Bowl | 55 |
| Figure A.13 | Sample input image and output level: Chimneys | 56 |
| Figure A.14 | Sample input image and output level: Water Wheel | 57 |
| Figure A.15 | Sample input image and output level: Combine Harvester | 58 |
| Figure A.16 | Sample input image and output level: Inclined Plane | 59 |
| Figure A.17 | Sample input image and output level: Catamaran | 60 |
| Figure A.18 | Sample input image and output level: Ferry | 61 |
| Figure A.19 | Sample input image and output level: Galleon | 62 |
| Figure A.20 | Sample input image and output level: Rowboat | 63 |

Abstract

Procedural Content Generation (PCG) attempts to define an automated process for designing game content. Angry Birds represents a physics based game in which interesting game play comes from the challenge and/or fun associated with destroying structures using various types of ‘angry birds’. This thesis proposes and demonstrates an approach to automating the design of the structures using a database of source objects. The hypothesis is that the resulting structures will retain some properties that remind the game player of the original object, making it ‘fun’. On the other hand, these structures also tend to be more varied than those produced using current approaches to PCG for the Angry Birds game. The approach is demonstrated using a database of 20 images, resulting in content that we believe improves on the longevity and enjoyment of the game relative to current practice.

Chapter 1

Introduction

Procedural Content Generation (PCG) for games attempts to build game content through some automated algorithm as opposed to having to design everything a priori. This means that the game designer does not need to prescribe absolutely all the content, but can concentrate on establishing the overall properties of the game. PCG has been extensively applied to automating the process of creating realistic visual properties in video games, such as vegetation (for example, the SpeedTree algorithm by Interactive Data Visualization Inc.). In this thesis, I am interested in the case of PCG for designing structural properties of the physics style game ‘Angry Birds’.

PCG for this purpose might be characterized in terms of constructive, generate-and-test or search based methods (although several other characterizations might be considered [7]). Constructive frameworks create content once, and therefore emphasize producing the same content each time. Moreover, any content generated this way must be ‘correct’. Generate-and-test schemes iterate between generating content and testing for the satisfaction of specific test criteria. Depending on which tests are satisfied/not satisfied all or part of the generated content is modified, with the process terminating once satisfying content is identified. Finally, search based methods imply that feedback on the content takes the form of a real-valued performance vector that can be used to direct an iterative search process for appropriate content.

In this work, I will be interested in the specific case of generating the structural content for targets in the Angry Birds physics style of game. An open source game engine already exists for this game,¹ where this has been used to demonstrate various approaches to building structural content or defining strategies for ‘angry bird’ deployment. Past research has concentrated on the design of fitness functions for evolutionary computation (EC), so that individuals from the evolutionary method

¹<https://aibirds.org>

design structures that are in some way guaranteed to be stable [1], [2]. To do so, such an approach need their own physics simulation in order to test the many configurations of structure that the EC method produces. Refinements to this approach include developing probabilistic models for which block (structures) are more likely to be stable [5] and assuming specific build heuristics for guaranteeing the stability of the resulting structures [6].

My approach will build on the previous research by assuming heuristics to ensure the stability of structures. However, rather than use some random process to define the build for structures, I will make use of silhouettes derived from real images to represent the outline of the structure. I will then fill the silhouette with blocks to provide an initial build. This can then be refined using the heuristics to provide stability to the resulting structures, i.e. introduce rectangular horizontal blocks to overhangs. Finally, I provide a heuristic for automatically placing ‘pigs’ within the structure. This represents an example of a constructive approach, although the same source image might give multiple ‘builds’ through changing the basic building block, or changing the aspect ratio.

This thesis develops by presenting background to the game play and basic history of the Angry Birds game, Chapter 2. The approach I adopt for providing automatic content generation from image data is detailed in Chapter 3. Chapter 4 provides the detailed description for the algorithm I adopt for building structures from the image. Chapter 5 provides an summary to the various outputs produced by the content generator for a specific source object, with thesis Conclusions and Future work appearing in Chapter 6. For completeness Appendix ?? provides a complete summary of the source object images and resulting Angry Birds structures.

Chapter 2

Background to the Angry Birds Game

Angry Birds is an example of a physics based puzzle game defined in terms of a 2-D scene and played on a wide range of mobile and personal computing devices. The game was originally created by Rovio Entertainment¹ and initially released in 2009. The original objective of the game is to have birds rescue their eggs from green-coloured pigs. The pigs are positioned in a structure constructed out of blocks, with different coloured blocks having different properties. During game play, the user defines how a sling shot is aimed/parameterized to fire the birds at the structures. Different coloured birds have different ‘strengths’, and as the player progresses through levels of the game, more complex structures are encountered, and a greater range of bird types are made available.

More points are awarded to the player for successfully killing the pigs and destroy as much of the structure using as few of the birds as possible. Most birds also have properties that can be triggered during flight. For example, the black and white birds have the ability to make explosions. The work of this thesis will assume the open source ‘Angry Birds AI’ game engine.² This was originally designed for testing AI agents that attempted to design good strategies for launching the angry birds. As such the AI agent would not have prior knowledge of the structures. The ‘Angry Birds AI’ game engine has been used for competitions at international AI and computer gaming conferences since 2012. I will instead be interested in developing content for the structures in which pigs are placed, or level generation³. Ultimately, the goal is to provide automatic content that is ‘fun and challenging’ to play.

The hypothesis of my approach is that by using silhouettes extracted from real-images, I can provide content that is both challenging and fun. There will be lots of repeating patterns, and positions for pigs, and there will be the potential for structures

¹<https://www.rovio.com>

²<https://aibirds.org>

³<https://aibirds.org/other-events/level-generation-competition.html>

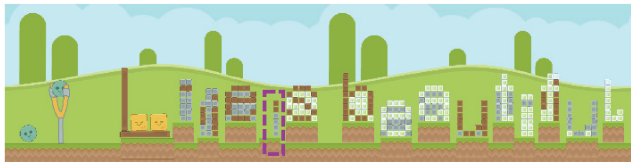


Figure 2.1: A structure built using the PCG of Jiang et al. [4].



Figure 2.2: A structure built using the PCG of Ferreira et al. [2].

to remind the user of the underlying object. During the evaluation of my framework for automatically building structures I will consider how much of the structure is stable, as well as how much of the structure provides a resemblance to the original object.

The closest previous research to the framework proposed in this thesis is that of Jiang et al. [4]. They provide an algorithm for configuring blocks to represent the characters of the alphabet. Their level generator then assembles sequences of characters (and therefore blocks) using a database of quotes. Figure 2.1 provides an example of levels generated using this approach. I note that these structures are not particularly complex, and generally have a horizontal orientation.

From the perspective of creating more complex builds, the most recent work of Ferreira et al. is capable of constructing both high and wide aspect ratio structures [2]. However, as can be seen from Figure 2.2 these structures lack a theme, and are generally unrelated to structures that might correspond to every-day objects.

Chapter 3

Image Capture and Pre-processing

We start by selecting an image that has the potential to be a good candidate for constructing a level. This potential is primarily determined by the shape of the outline of the object represented by the image, as the content, lighting, coloring, resolution, size and even format is not of any concern of us for the purposes of generating a level. This is because the only use of the input image, the raster image, for us is to convert the input image into a vector image, specifically a Scalable Vector Graphics (SVG) image, before executing the code that actually generates a Science Birds structure. Since this code operates on an SVG image, specifically an SVG image with a polygon element, all the operation that we do before feeding an SVG image that contains a polygon element to the code can be simply understood as the "preprocessing stage" for constructing a Science Birds structure. Nevertheless, in order to start with a raster image and attain a Science Birds structure, all the steps in the preprocessing stage are necessary. These steps are listed in Figure 3.1.

Some of the steps in the flowchart are explained in detail below:

1. Converting a raster image to black-and-white: The reason for this is actually clear. We have stated that the content, lighting, coloring, resolution size or format of the image is of no concern to us, as we have stated that the only use of the image is the outline of the structure represented in that image. We have also stated that the raster image, for the purposes of generating a Science Birds structure is of no use to us, since it is not possible (or, straightforward) to obtain the outline data of the image from the raster image. Since that's the case, content, lighting and similar are simply redundant information for all of our purposes. Hence, in order to have an easier and more precise conversion from raster to vector, converting a raster image to black-and-white is a useful first step.

For example, if the given input image is the image in Figure 3.2, after the end

Figure 3.1: Flowchart demonstrating the process of vectorizing the raster image

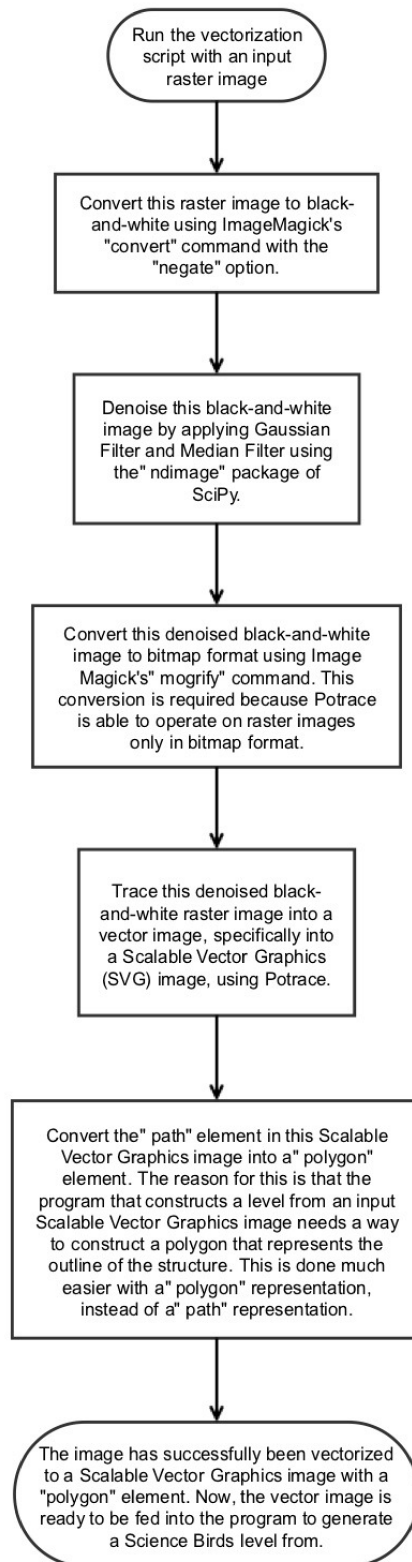


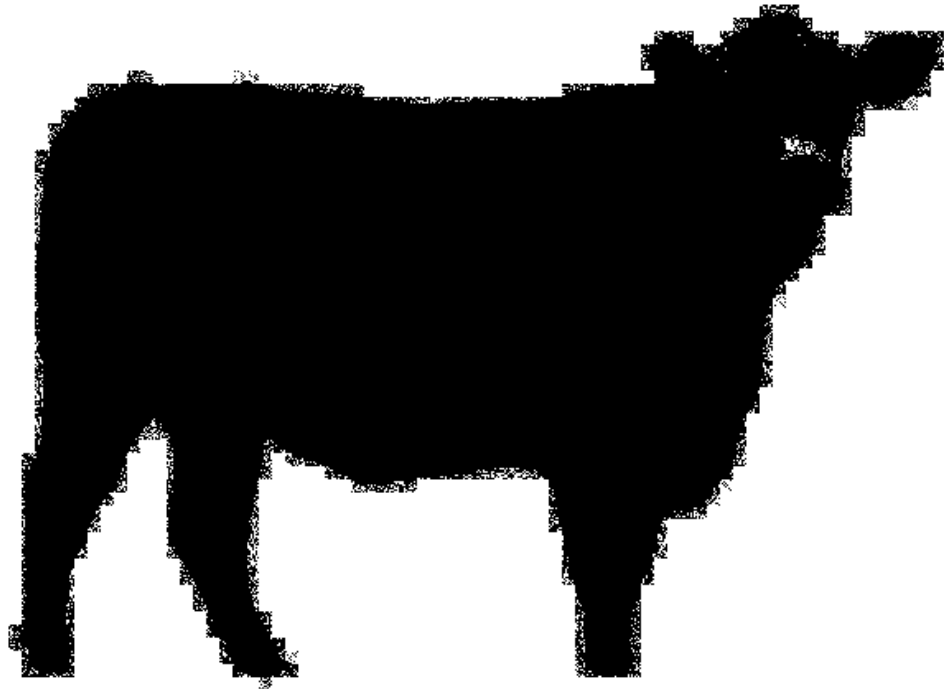
Figure 3.2: Input cow image



of this step, the image in Figure 3.3 will form.

2. Denoising this black-and-white image: Although converting a raster image to black-and-white comes a long way for vectorizing that image, the black-and-white image after the conversion is in fact, noisy. That is, the image is in black-and-white and for the most part, represents the shape of the structure that is depicted in the original raster image. However, there is some granularity, some noise, on the edges of the structure after the conversion. This is, of course something that is not desirable, since this only makes it harder to trace this image. That is, tracing the image in this state can result in multiple, irrelevant curves in the output, instead of a single curve that represents the structure. This is not something we want, since the irrelevant curves will simply result in erroneous structure constructions. Hence, in order to remove this unwanted effect, we simply denoise the resulting black-and-white image and hence, we obtain a smooth black-and-white image that is much more suitable for producing a fine and sharp vector image.

Figure 3.3: Cow image converted to black-and-white



After this step, the image in Figure 3.4 is formed.

3. "Tracing" this denoised, black-and-white image in order to obtain an SVG image: Theoretically, the raster image can be traced into a vector image using any method. Specifically, we are using the open source program named Potrace, written by Peter Selinger, for this task. One of the rather important reasons for converting the original raster image to black-and-white, instead of directly tracing it (that is, creating a vector image out of it) is that the tracer (vectorizer) software that we use, Potrace, works only on black-and-white images. That is, the behavior of Potrace for non black-and-white images is undefined. Hence, we opt to use a black-and-white image as the input for Potrace. That is the main reason that we convert the original raster image into black-and-white, instead of feeding it to Potrace as is.

After this step, the image in Figure 3.5 is formed.

4. Potrace traces the raster image into a vector image successfully. However, in the output, it uses SVG path elements. For general purposes, this is perfectly

Figure 3.4: Black-and-white cow image after being denoised



fine. However, the main reason that we need the SVG is in order to be able to cleanly determine the outlines of the structure represented by the raster image. The reason for doing this is as follows: When the image is in one of the raster formats, determining the outline of the structure represented by this raster image is not very straightforward. However, since all we need for the purpose of generating a Science Birds structure is the outline of the structure represented in the raster image, we need a way to attain the information about the outline of the structure represented in the raster image. This is where the SVG, specifically an SVG with a polygon element, comes into play. Normally raster images represent an image as a collection of pixels. Basically, in a raster image, there is no notion of a shape. That is, in a raster image, mostly the image is represented as a collection of independent picture elements (pixels). Hence, in a raster image, it is not straightforward at all to determine what sort of shape is being represented by this image. In other words, there is no proper notion of a "shape" in a raster image.

Figure 3.5: Black-and-white cow image after being denoised and converted to Scalable Vector Graphics

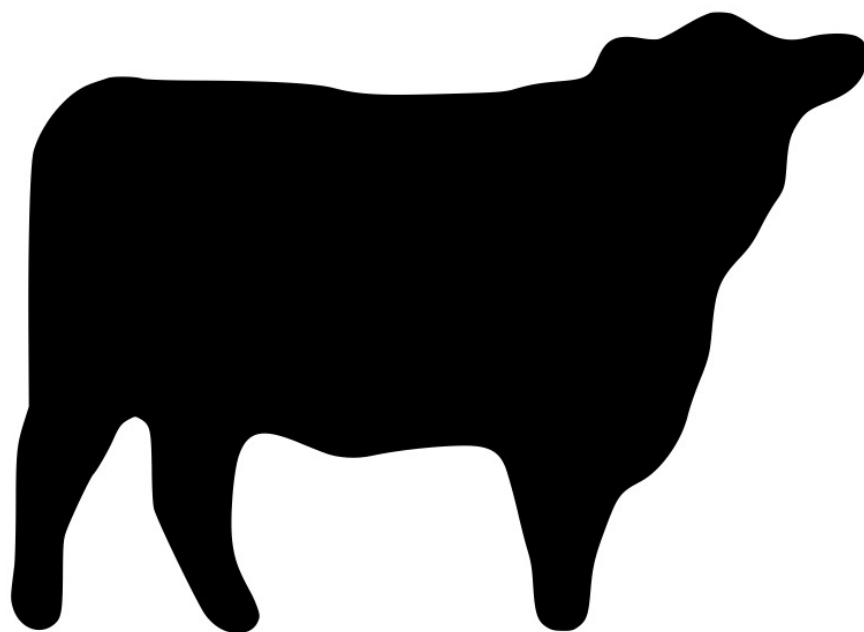
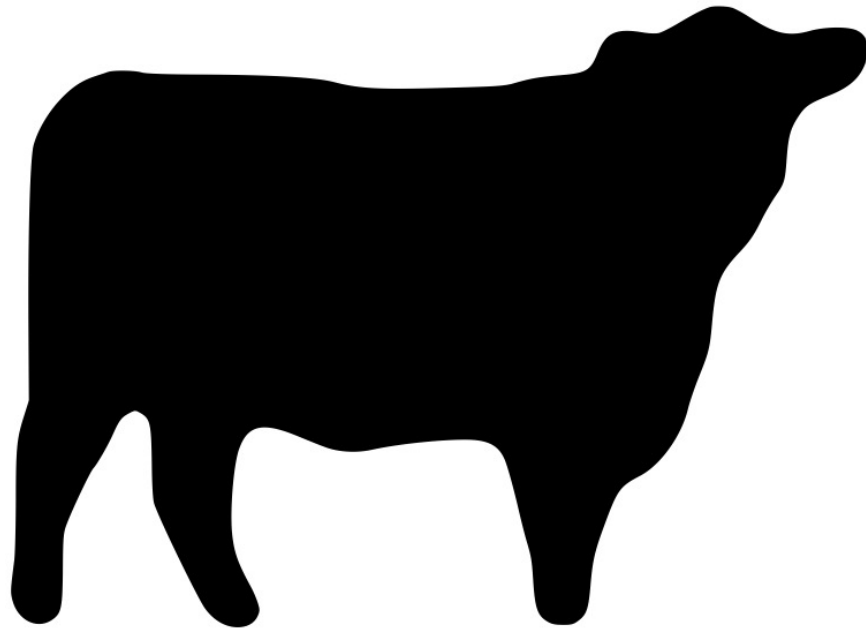


Figure 3.6: Vectorized cow image with a ‘polygon’ element, instead of a ‘path’ element



After this step, the image in Figure 3.6 is formed.

On the other hand, vector image formats are the opposite of raster image formats. While raster image formats usually represent an image as an unrelated collection of picture elements, a vector image format, on the other hand, represents an image as a collection of shapes. It is this fundamental difference in the approach of image representation that differentiates the raster and the vector image formats. Since a vector format represents an image as a collection of shapes, a vector image is not as prone to alterations in the image quality as a result of altering the viewing point of the image. For example, a vector image is almost immune to degradation in quality when zooming in the image. It simply shows the zoomed in detail of the image as represented among the shape declarations of the vector image. It is precisely this characteristic of vector images, specifically SVG, that catches our attention. As we have stated before, for the purpose of generating a Science Birds level from an input

raster image, all the information that we need from that raster image is simply the shape, that is, the outline, that this raster image represents. This outline information is best conveyed to the level generator with an SVG image that represents the image with an SVG polygon element. Hence, this is precisely the reason that we need an SVG image with a polygon element in order to cleanly attain the information about the outline of the structure represented in the raster image.

Chapter 4

Algorithm

After obtaining the Scalable Vector Graphics (SVG) image which as detailed in Chapter 3 is produced by:

1. Converting the raster image into black-and-white
2. Denoising it
3. Tracing it

we feed this SVG image into the main script. In the main script, the following happens:

1. The configuration file is read in order to determine where should the output level file be written to. The output level file is an XML file which contains a description of a Science Birds level.

4.1

Note that Science Birds developers have opted not to be very strict about the validity of XML in the level files. That is, some elements in the level actually turn the XML into invalid XML. Nevertheless, the level is still openable and it is still playable. Some examples to these invalid elements would be the elements Camera and Slingshot. Although these elements do not have a separate closing element, they don't end with `</>` as they should be in valid XML. Instead, they end with `>`, just like a self-closing tag in HTML. However, precisely speaking, a Science Birds level file is not an HTML file, it is an XML file. Hence, the aforementioned non-conforming elements should have either:

- (a) Had a closing tag.

or

Listing 4.1: A Science Birds level template

```
<?xml version="1.0" encoding="utf-8"?>
<Level>
  <Camera x="" y="" minWidth="" maxWidth="">
  <Birds>
    <Bird type="" />
    ...
  </Birds>
  <Slingshot x="" y="" >
  <GameObjects>
    <Block type="" material="" x="" y="" rotation="" />
    ...
    <Pig type="" x="" y="" rotation="" />
    ...
    <Platform type="" x="" y="" scaleX="" scaleY="" />
    ...
    <TNT type="" x="" y="" rotation="" />
    ...
  </GameObjects>
</Level>
```

- (b) Ended the single tag elements with `>`, instead of `/>`.

in order to ensure that the XML files of Science Birds levels are, indeed, valid XML files.

The sample level file structure scheme above demonstrates what a generic level would be similar to. However, it doesn't go into detail of explaining what these elements are, what is their use or how do their attributes alter or enhance their behavior. In order to gain a solid understanding of the level scheme of Science Birds, we need to understand the individual elements, and the attributes of each of these individual elements that make up a Science Birds level. To do so, we need to examine the sample Science Birds XML level structure step-by-step, examining one line at each step:

- (a) `<?xml version="1.0" encoding="utf-8"?>`

This is simply a generic XML file declaration. Without this element, we wouldn't be able to claim that a Science Birds level file is indeed, an XML file, even though it might be composed of elements within angular brackets and the file extension in a Science Birds level filename is `.xml`.

- (b) `<Level>`

This is the top-level container element that contains all the other elements that are necessary to be able to construct a complete, working Science Birds level. This is actually a container element. It is the only top-level element apart from the XML declaration line.

- (c) `<Camera>`

This element exists to describe where should be the view be when the level starts, and what how large should it be.

- (d) `<Birds>`

This is the element that contains all `<Bird>` elements. In other words, this is a container element for the `<Bird>` elements.

- (e) `<Bird>`

This is the element that adds a bird to the list of birds that can be used in this level. There are different types of birds. These types are:

i. BirdWhite

BirdWhite is a bird that is comparatively larger than the rest of the birds. The large size of this bird gives the advantage of being able to knock down more blocks due to:

A. Larger impact area.

B. Higher impact force due to higher momentum.

Figure 4.1: White bird



ii. BirdBlue

BirdBlue is a regular bird that is smaller than BirdWhite.

Figure 4.2: Blue bird



iii. BirdYellow

BirdYellow is around the same size with BirdBlue, only that it is of yellow color.

Figure 4.3: Yellow bird



iv. BirdRed

BirdRed is similar to BirdBlue and BirdYellow, only that it is of red color.

Figure 4.4: Red bird



v. BirdBlack

BirdBlack is very similar to BirdWhite in terms of size. However, BirdBlack has one property that is not observed in any of the other birds. It has the ability to explode. Upon making the first contact with any kind of surface, be it the ground, a block, a platform, a pig or a TNT block, BirdBlack becomes "rigged to explode". This "rigged to explode" state is expressed by the BirdBlack's body flashing red intermittently. That is, its body is covered with a red hue, and then it becomes normal. This state repeats a couple times a second until the bird explodes, which takes a couple seconds. Upon explosion, any movable thing that is in the immediate surrounding of the BirdBlack is pushed away from BirdBlack. This can help with knocking down more blocks and eliminating more pigs (that is, enemies), if the BirdBlack is launched in a manner that the explosion will happen in a strategically beneficial place which will allow these.

Figure 4.5: Black bird



(f) `<Slingshot>`

The slingshot is used to launch the birds into a trajectory in order to knock down the structures and to eliminate the enemies that are present on the level. Its attributes `x` and `y` are used to express the location on the two-dimensional level platform that the slingshot should be placed to.

(g) `<GameObjects>`

This is a container element that is used to contain every single element in a Science Birds level, apart from the birds and the slingshot. That is, this element is used to contain all elements that a bird is able to make contact with, with or without effect. Specifically, it can contain `<Block>`, `<Platform>`, `<Pig>` and `<TNT>` elements.

(h) `<Block>`

This is the main element that is used to construct structures. Basically,

a Science Birds structure is an aggregation of <Block> elements. Technically, <Platform> elements and <TNT> elements can be used as part of a structure as well, along with using a <Pig> element as the enemy. However, in this programmatic content generator, a Science Birds structure consists only of <Block> elements.

There are precisely thirteen different shapes of blocks. Each block type has four integrity conditions. That is, when the blocks hadn't taken any damage, they are in perfect condition. As they take more damage, their condition changes. Finally, after enough damage, a block disappears and the player gets points from this.

Of the thirteen types of blocks, twelve of them has variations in all three materials, namely "ice", "wood" and "stone". Only one of them can be drawn in only one material, which is Ice Square.

Each of the four different integrity conditions of the blocks are represented by a different sprite for each block type. Hence, the total number of sprites for all blocks is as follows:

$$12 * 3 * 4 + 4 = 148$$

The blocks, except the single material type block Ice Square, are the following:

- Small Circle
- Circle
- Triangle
- Triangle with Hole
- Small Square
- Square
- Square with Hole
- Tiny Rectangle
- Small Rectangle
- Medium Rectangle
- Big Rectangle

- Fat Rectangle

So, the first two attributes of the `<Block>` element are now clear. The `type` attribute is used to indicate which block type this block should stand for, where those block types are listed above. On the other hand, the `material` attribute is used to specify which material this block should be of. The possible materials are:

- Ice
- Wood
- Stone

The remaining attributes are `x`, `y` and `rotation`. These have the following respective purposes:

`x`: This attribute specifies the location of this block on the X coordinate. The origin is the top-left location and the X coordinate increases towards right.

`y`: This attribute specifies the location of this block on the Y coordinate. The origin is the top-left location and the Y coordinate increases towards down.

`rotation`: This attribute specifies the angle which this block is placed according to.

(i) `<Pig>`

This element specifies a pig. The attributes are as follows:

- `type`: This attribute specifies the type of the pig. Possible values are:
 - BasicBig
 - BasicMedium
 - BasicSmall
- `x`: This attribute specifies the location of this pig on the X coordinate.
- `y`: This attribute specifies the location of this pig on the Y coordinate.
- `rotation`: This attribute specifies the angle which this pig is placed according to.

(j) <TNT>

This element represents a TNT block. A TNT block is a block that is rigged to blow. Upon activation, a countdown for that TNT block starts and at the end of the countdown, the TNT block explodes. A TNT block becomes activated whenever another block hits it. The explosion of a TNT block destroys blocks and eliminates enemies in the surrounding of the TNT block, within a certain diameter.

2. After the configuration file has been read, the following parameters are extracted from it:

- (a) LevelPath: This parameter specifies the directory where the Science Birds levels should be written to, in order to be able to play them upon launching Science Birds.
- (b) PrimaryBlock: This parameter specifies the block that is used to construct the structure. A structure in Science Birds Programmatic Content Generator is constructed using two types of blocks: primary block and platform block. primary block is the block that is structure is initially constructed from. That is, the structure initially solely consists of primary blocks.
- (c) PlatformBlock: This parameter specifies the block that is used to support the primary blocks that have nothing under them to prevent them from falling. After the structure is initially constructed solely by using primary blocks, in most cases, there are some primary blocks that happens to have nothing under them. Had these blocks be left in this state, they would simply start free falling upon level start. This is, of course, not a desirable thing, since in such case, the player would get undeserved points due to elimination of some blocks because of free falling. Also, the free falling blocks might result in imbalances in the structure and hence, result in toppling of the structure as well. Because of these reasons, in order to have a level with a stable and playable structure, we must make sure that there shouldn't be any blocks without anything supporting them below. Hence, the program determines where exactly to place these platforms and places

these platform blocks accordingly. The details of the platform insertion procedure is examined in detail in the latter parts of this dissertation.

- (d) `NumberOfPrimaryBlocksOnXAxis`: This parameter specifies how many primary blocks should be placed to X axis. Initially, one might think that this should be determined by dividing the width of the structure to the width of the primary block. This is, in general, an accurate and sensible approach. However, there is a caveat: If the structure too wide, and the primary block's width is not wide enough, there will be a lot of blocks on the X axis, and as a result of this, on the Y axis as well. Hence, the structure will be composed of too many blocks. Similarly, if the structure is rather thin, and the primary block's width is wide relative to the width of the primary block, there will be too few blocks on the X axis, and as a result of this, on the Y axis as well. Hence, the structure will be composed of too few blocks.

Neither of these situations desirable. If the structure happens to be constructed of too many blocks, there will be three problems:

- (a) The rendering of these blocks will take more resources and hence, it might lead to a degraded experience.
- (b) Since there will be too many blocks, it will take more effort to knock down the structure. Similarly for eliminating all the pigs in the level. For example, it might require more birds to knock down the structure or eliminate all the pigs. In most cases, this is something that degrades the experience. Hence, having more than necessary amount of blocks for a structure is something to be best avoided.

In the same fashion, having too few blocks in a structure is an undesirable thing as well. If the structure is composed of too few blocks, it most likely will be too thin and short. In such case, most likely there won't be enough space in the structure to insert platforms and place pigs (enemies). There cannot be a playable level without any pigs, since the goal of each level is to eliminate every single pig in that level. For these reasons, having a level with a too thin structure is not something that we would like either.

So, we understood that if we decide on the number of primary blocks on the X axis by dividing the width of the structure to the width of the primary block, there is room for numerous problems. Hence, dividing the width of the structure to the width of the primary block is not the optimal method to decide on the number of primary blocks on the X axis and implicitly, the number of blocks of the whole structure.

Figure 4.6: Example of a structure with inadequate number of blocks for pig insertion



What then, might be a better way to decide on the number of primary blocks on the X axis? As it is currently implemented, deciding on it manually, that is, using a parameter seems to be the best way now. One point to clarify here is that if we decide on the number of primary blocks on the X axis manually, instead of dividing the width of the structure to the width of the primary block, then how are we going to decide on the number of primary blocks on the Y axis? Should we use another parameter for that as well? Well, the answer is no, because doing that would simply remove the dependence between the structure's width of the X axis and on the Y axis. Removing this would result in the created structure being out of shape. Hence, in order not to cause such

an issue, we instead proportionate the dimensions of the primary block to the dimensions required by the specified number of primary blocks on the X axis. Precisely, we do this as follows:

- (a) Divide the width of the structure to the specified number of primary blocks on the X axis.
- (b) Divide this result to the actual width of the primary block. The result of this gives us the block factor.
- (c) Multiply the block factor with the height of the primary block.
- (d) Divide the structure height by the result of the calculation done at step 3. The result will give the number of blocks that should be on the Y axis.

3. After reading the arguments, we read the SVG file in order to determine the shape and boundaries of the structure that we are about to construct. This is done as follows:

- (a) Read the SVG file.
- (b) Find the SVG <polygon> element in it.
- (c) Get the list of points in that <polygon> element.
- (d) Using this list of points, construct a Polygon object. This is done with the help of a library called Shapely.
- (e) Rotate this polygon object 180 degrees (upside-down). The reason for this step is that Potrace generates the SVG image in an upside-down form. In order to properly use the image to construct a Science Birds structure, we need to access the SVG image's proper (upright) version.

4. After reading the SVG file and constructing a polygon from it, we initialize a Structure object. A structure object is an object that represents a Science Birds structure. It represents every aspect of it. Some of these are:

- (a) The original polygon shape that is read from the SVG file.
- (b) All of the arguments:

- i. The level path.
 - ii. The primary block type.
 - iii. The platform block type.
 - iv. Number of primary blocks on X axis.
- (c) Number of primary blocks required to cover pig width: In this PCG, the pigs are placed by removing some primary blocks and placing the pig to the bottom center of the cavity as a result of this removal. Hence, the width created by removing the primary blocks should be at least as long as the pig's width. Otherwise, the pig simply wouldn't fit in the cavity created by removing the pigs. Hence, in order to determine this width, we make a calculation to determine the number of primary blocks to remove, so that the width created by this removal will at be at least equal to the width of a pig.
- (d) Number of primary blocks required to cover pig height: This is the same concept as "number of primary blocks required to cover pig width".
- (e) Primary block factor: This is the concept that we have mentioned about in the definition of "NumberOfPrimaryBlocksOnXAxis" parameter. To remember what we have stated there, the primary block factor is a ratio that dictates the factor the primary block dimensions need to be multiplied with in order to cover the structure width and the structure height. Note that the structure width and the structure height simply refers to the width and height of the rectangle that would contain the structure that is expressed in the SVG file. In other words, the "width" and "height" refer to the containing rectangle of the structure that is expressed in the SVG file.
- (f) Factored primary block width: This is the distance that is calculated by multiplying the primary block width with the primary block factor.
- (g) Factored primary block height: This is the distance that is calculated by multiplying the primary block height with the primary block factor.
- (h) Number of primary blocks on X axis: This is calculated by dividing the structure width by the factored primary block width.

- (i) Number of primary blocks on Y axis: This is calculated by dividing the structure height by the factored primary block height.
- (j) Original blocks: This is the matrix (list of lists) that holds the blocks of the structure location by location. As you can realize, this is not something that has been passed to the Structure class' constructor. That means this must be computed within the Structure class. Precisely, this is done in the `get_blocks` method of the Structure class. The algorithm is basically as follows:
 - i. The structure shape that is obtained from the input SVG is partitioned into tiles, where the length and width of the tiles are dictated by the parameters `num_primary_blocks_on_x_axis` and `num_primary_blocks_on_y_axis`.
 - ii. Each tile is composed of:
 - A. Either only by a structure.
 - B. Or only by void.
 - C. Or both some structure and some void.

If the area of the structure within that tile is greater than or equal to half of the area of a tile, we decide that there should be a primary block inserted for this location. Otherwise, we leave this tile empty. We repeat this procedure for each tile and in the end, we obtain a list of lists, that contains True or False values, indicating that there should be a block in that location or not.
- (k) Blocks: This parameter is very similar to original blocks parameter, with the only difference being that this is the version of original blocks that is transposed and inverted. The original blocks parameter starts from top-left and continues towards bottom-right, as this is the natural procession order in SVG (and most image formats). However, for the purposes of constructing a Science Birds structure out of a matrix of blocks, the most practical way is to start from bottom-left and go towards top-right. The reason is starting from the ground and building a column, then proceeding to the next column and building it, until all columns have been finished.
- (l) Platforms: The platforms are obtained with the following algorithm:

- i. Determine every block which has nothing underneath it.
- ii. Determine the row number of each such block and add this row number to a set which contains the row indices for platforms to be inserted under.
- iii. Repeat this for every block on the structure.

These steps are for determining the necessary platforms. That is, they are for determining the rows to place platforms which are necessary in order to prevent blocks without anything under to free fall upon level start. In addition to the necessary platforms, there are also extra platforms. Extra platforms are the platforms inserted to the structure not because of preventing blocks from free falling, but to create places to be able to insert more pigs. The algorithm to insert extra platforms is as follows:

- i. Determine the topmost and bottommost platform.
- ii. Starting from the topmost platform and going above, insert an extra platform every N th row, where N is equal to the number of primary blocks required to cover pig height.
- iii. Similar to the previous step, starting from the bottommost platform and going below, insert an extra platform every N th row, where N is equal to the number of primary blocks required to cover pig height.

- (m) Platform blocks: In the previous step, we have determined the row indices where the platforms should be placed under. However, we still don't have the information about which lateral distance each platform block should be centered at. This is something that needs to be calculated. The reason is that although the primary blocks have fixed places, the platform blocks have not. That is, unlike the primary blocks, the platform blocks have lateral distances that can vary for each platform. Hence, the lateral distance of each platform block needs to be calculated individually.

The reason for this is the following: Platform blocks are placed according to the placement of the primary blocks which the platform is located under. That is, the main reason for placing platform blocks is to support the primary blocks that have nothing underneath them. That is, the main

reason is to prevent primary blocks without anything underneath them from free falling. Because of this, if a row is not filled with primary blocks, that is, if the primary blocks in a row does not start at the left edge of the shape's containing rectangle, and end at the end at the right edge of the shape's containing rectangle, then the distances of the platform blocks under that row is not regular, and they need to be custom calculated. The reason is that in such scenario, the platform blocks start and somewhere in the middle of the row, and end at somewhere in the middle of the row, instead of starting at the beginning and ending at the beginning. The algorithm to calculate the platform block locations is as follows:

- i. Determine the leftmost primary block in the row which the platform is going to be placed under. We do this by determining the column index of the leftmost primary block in the row which the platform is going to be placed under.
- ii. Determine the rightmost primary block in the row which the platform is going to be placed under. We do this by determining the column index of the rightmost primary block in the row which the platform is going to be placed under.
- iii. Using the leftmost primary block index, rightmost primary block index, and the total number possible primary blocks in a row (which is obtained using the parameter named "number of primary blocks on X axis"), determine how many primary blocks there are between the leftmost primary block in the row and the rightmost primary block in the row.
- iv. Using the number of blocks between the leftmost primary block in the row and the rightmost primary block in the row and using the width of the platform block, calculate the number of platform blocks required to cover the distance under the number of blocks between the leftmost primary block in the row and the rightmost primary block in the row, using platform blocks. This will give us the number of platform blocks required for that row.
- v. Using the number of blocks between the leftmost primary block in the

row and the rightmost primary block in the row, compute the center of the region between the leftmost primary block in the row and the rightmost primary block in the row.

- vi. Starting from the center of the region between the leftmost primary block in the row and the rightmost primary block in the row, we start inserting platform blocks. Platform blocks are inserted in a continuous manner. That is, except the initial iteration, at each iteration two platform blocks are added to the left and right ends of the current state of the platform. At the initial iteration, either one or two platform blocks are inserted. If the calculated number of platform blocks are odd, one platform block is inserted right at the center of the distance between the leftmost primary block in the row and the rightmost primary block in the row. This platform block is inserted in a manner such that the center of that platform block will correspond to the center of this distance.

If, on the other hand, the calculated number of platform blocks are even, then there will be two blocks inserted at the initial iteration. The blocks will be inserted in such manner that the rightmost edge of the left platform block, and the leftmost edge of the right platform block will correspond to the center of the distance between the leftmost primary block in the row and the rightmost primary block in the row. The rest of the iterations are the same, two platform blocks are inserted at each iteration, one prepended to the left edge of the current platform, and the other appended to the right edge of the current platform.

5. Pig indices: Pigs are inserted under every single platform block, where the previous platform is at least "number of primary blocks required to cover pig height" rows below the platform that contains this platform block. The reason is straightforward: If the previous platform is not at least "number of primary blocks required to cover pig height", then the pig won't have enough vertical space to fit in. This will cause the structure to have imbalances and to knock over.

Another thing to note is that every pig is represented by only a single block index. That is, every pig is represented by the primary block's location that is at the center of the platform block that the pig is located under. Then, using this index, the pig is inserted to the center of this gap that is created by removing the blocks. The precise steps followed for block removal and pig insertion are as follows:

- (a) After a platform block is determined to be suitable for inserting a pig under, the index of the primary block that is right under and at the center of this platform block is determined. Index is the top-center location of the pig to be inserted.
- (b) This block is removed.
- (c) If the "number of primary blocks required to cover pig width" variable is greater than one, then additional blocks need to be removed as well. Removal of additional blocks are performed as follows:

- i. Starting with the block that is at the immediate right of the center block, start removing the blocks by alternating between the block that is near (after) the rightmost edge of the gaps, and at the leftmost edge of the gaps.
- ii. Repeat this "number of primary blocks required to cover pig height" times, going one row down at each step.

To give a concrete example, let's say that the "number of primary blocks required to cover pig width" is 5. Then, the index offsets of the primary blocks to be cleared are as follows:

0, 1, -1, 2, -2

The "index offset" refers to the block position that is relative to the index of the primary block that is right under and that corresponds to the center of a particular platform block.

Of course, this block removal is repeated as many times as the "number of primary blocks required to cover pig height", starting from the row that is immediately under the platform block and going downward.

- (d) After the block removal is completed, the pig is inserted. The exact location to insert the pig is calculated as follows:
- i. If the "number of primary blocks required to cover pig width" is an odd number, then the horizontal distance to place the pig is equal to the horizontal distance of the center of the primary block that is located right below the center of the platform block that this pig is being inserted under. If the "number of primary blocks required to cover pig width" is an even number, then we add half of the primary block's width to this distance to find the horizontal distance to insert the pig to.
 - ii. The vertical distance is calculated as follows:
 - A. Divide the "number of primary blocks required to cover pig height" by two. The division will result in a quotient and a remainder. Of course, if the "number of primary blocks required to cover pig height" is even, the remainder would be zero.
 - B. Subtract "quotient + 1" from the row index of the platform that contains the platform block which the pig is being placed under.
 - C. Calculate the height of the primary block row that corresponds to the resulting row index.
 - D. If the "number of primary blocks required to cover pig height" is an even number, subtract half of the primary block's height from this height.

This concludes the members of the Structure class.

Chapter 5

Example of the Build Process

This chapter walks through construction of a Science Birds level from a given raster image step-by-step. The aim is to let the reader to gain a better understanding of the level construction process.

Let's assume that the user of the programmatic content generator has obtained the following image:

Figure 5.1: The raw, unedited cow image



First, the user has to trim this image so that only the cow itself is remaining in the image. The details of accomplishing this is left to the user. A common method is to simply use an image editing software to cut out the cow from the whole image.

After the cow is obtained, the image becomes the following:

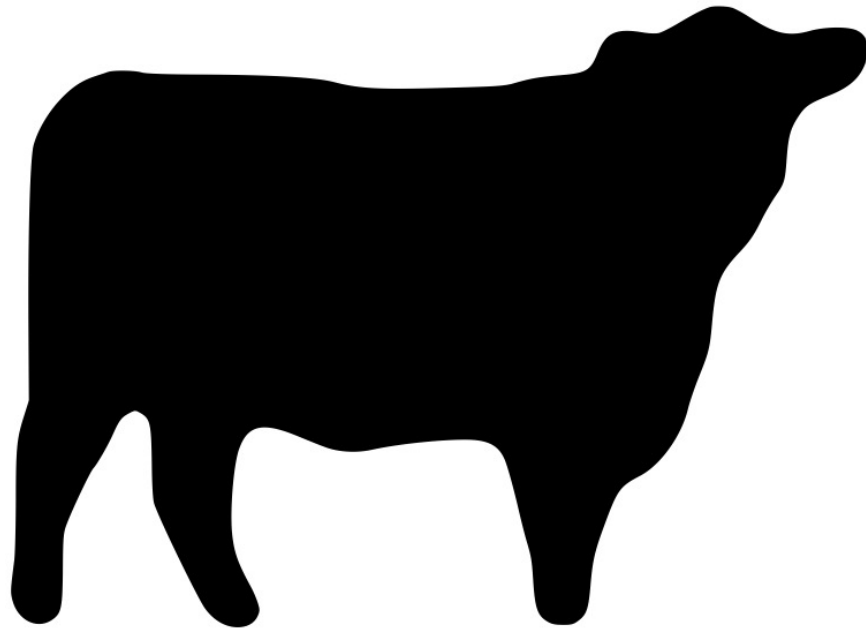
Figure 5.2: Edited cow image



After that, this image is converted to black-and-white, denoised and vectorized. The details of these processes are in chapter 3, hence they won't be repeated in this section. For further information, please refer to chapter 3.

After vectorization, the following image is obtained:

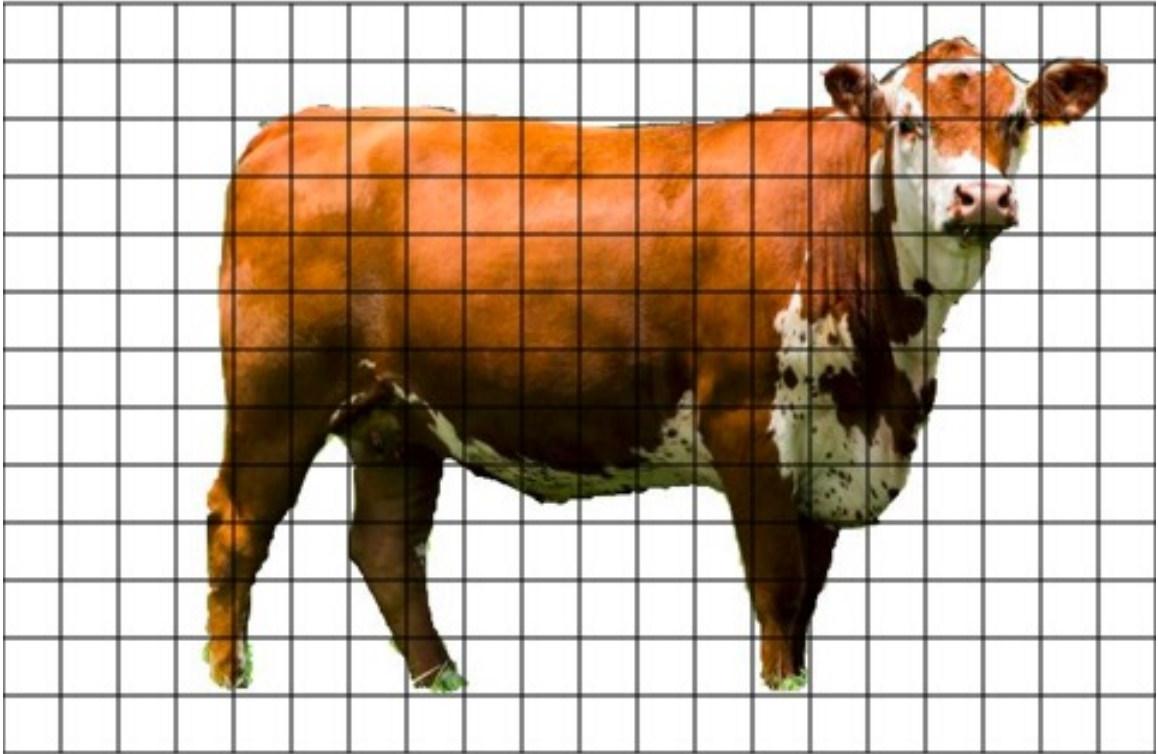
Figure 5.3: Vectorized cow image



This image represents the silhouette that is obtained from the input image. After that, this image is fed into the programmatic content generator for the actual level generation.

In the programmatic content generator, the image is partitioned into a mesh. The shapes of the individual loops (the identical components that compose the mesh) are the same as the shape of the primary block that is declared in the configuration file. The only difference between the two is the size of the loops. According to the size of the input image, without changing the aspect ratio, the size of the loop can be shrunken, expanded or kept the same. This is done in order to comply with the number of primary blocks to have on the X axis, which is again declared in the configuration file.

Figure 5.4: The mesh on the cow image



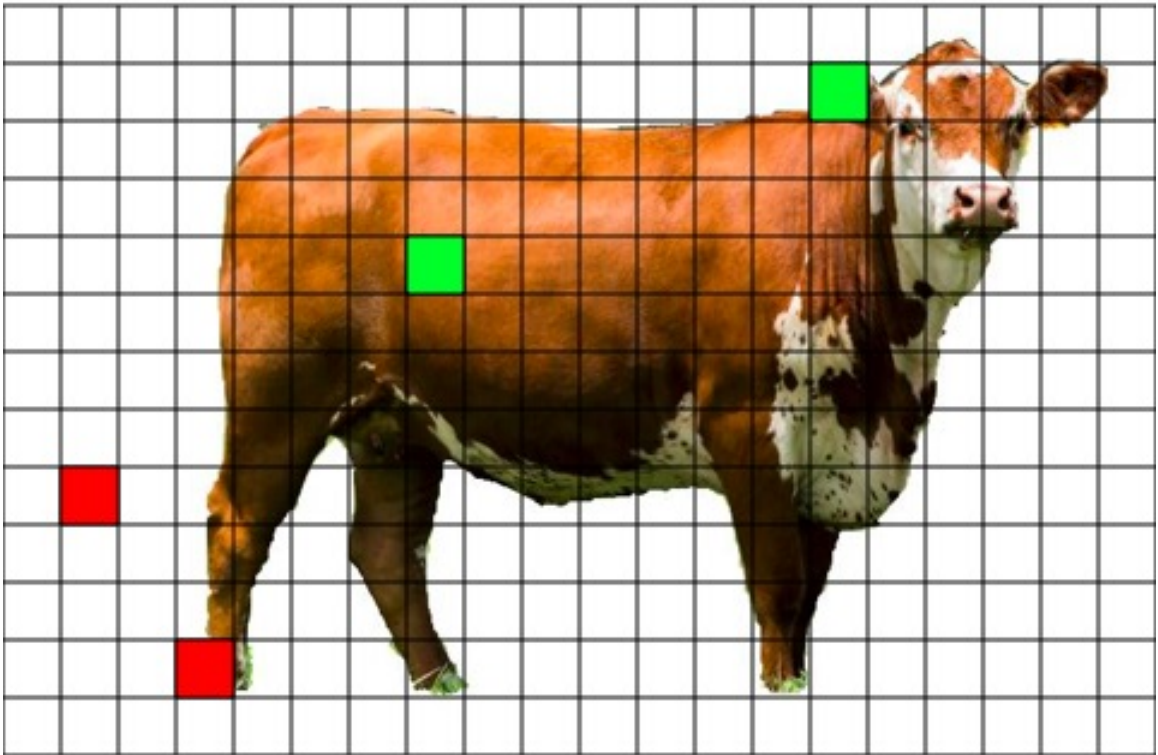
Since the primary block of our choice for this example has a square shape, the loops of the mesh are square shaped as well.

Each loop of the mesh is a place to potentially place a primary block. The decision process of whether or not to insert a primary block to a given loop is done as follows:

1. Calculate the area covered by a loop of the mesh. This computation needs to be done only once, since every loop of the mesh is identical.
2. Calculate the area within that mesh that belongs to the entity whose silhouette is being used to construct a structure. In this example, this entity is the cow.
3. If the area found in step 2 is greater than or equal to half of the area found in step 1, insert a primary block there. Otherwise, leave it empty.

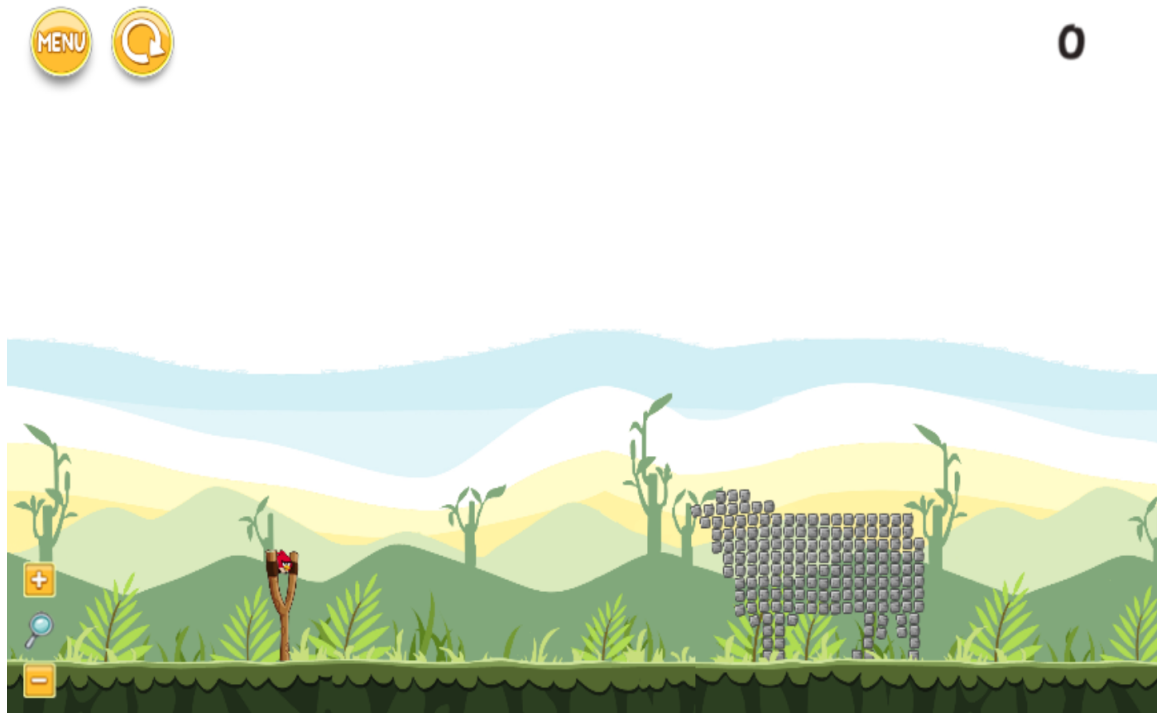
Below, four loops in the mesh of the cow are selected to demonstrate this. The green ones are the ones where a primary block will be inserted, and the red ones are the ones where a primary block will not be inserted.

Figure 5.5: Individual loops in the mesh that will contain a primary block and that won't



After this process is repeated for every loop in the mesh, we obtain a preliminary structure that consists of only the primary blocks. In this example, this would be as follows:

Figure 5.6: Generated structure with only primary blocks



As a side note, the reason for the flipping of the image with respect to Y axis is Potrace. As explained in chapter 4, Potrace turns the image upside down tracing it. In other words, it flips the image with respect to X axis from the vertical center of the image. Similar to this, Potrace flips the image with respect to Y axis from the horizontal center of the image. Although we take measures to re-flip the image to upright position, we don't take measures to re-flip to image to the original horizontal orientation.

After the primary blocks have been inserted, it is time to insert the platform blocks. As you can observe, some primary blocks do not have anything underneath them. That is, some of them just 'hang' in the air. However, the truth is, they do not hang in the air. When the level starts, as a result of the gravity in the physics engine of the game, they fall down. Hence, we need to prevent this from happening.

In order to prevent this from happening, we use the following algorithm:

1. Find out which primary blocks have nothing underneath them.
2. Get the row index (the order of the row starting from the ground and going upward) and put this row index into a set.

3. After this has been repeated for every block, insert platform blocks below every row that is indicated in the set that is created in the previous step.

After the insertion of the platform blocks, the structure becomes the following:

Figure 5.7: Generated structure with primary blocks and the inserted platform blocks



After the insertion of the blocks, we insert the enemies, which are the pigs. The algorithm for it is as follows:

1. Find out how many primary blocks each pig covers vertically. This is done simply by dividing the height of a pig by the height of the primary block. This will give us the number of primary blocks to cover pig height. This will be done only once.
2. Find out how many primary blocks each pig covers horizontally. This is done simply by dividing the width of a pig by the width of the primary block. This will give us the number of primary blocks to cover pig width. This will be done only once.
3. Find the platform block rows where the next platform block row below is away by at least the number of primary blocks to cover pig height.

4. For each platform block of each such row, starting from the center of the platform block, start removing primary blocks alternating right and left. Stop removing primary blocks until the number of primary blocks to cover pig width blocks have been removed.
5. Repeat the previous step downwards for the number of primary blocks to cover the pig's height.
6. Insert a pig to the center of the space created.

After inserting the pigs, the structure is as follows:

Figure 5.8: Generated structure with primary blocks, the inserted platform blocks and pigs



Note that because there is insufficient vertical height between any other rectangular block, we only find two pig positions in this case. Further refinements could include adding extra 'depth' between rectangular blocks.

Chapter 6

Conclusion

The general goal of PCG is to provide new content through an automated process. This has the potential to improve the longevity of the game itself as the content is never explicitly the ‘same’ and/or there is never an ‘end’ to the game. Moreover, it also implies that the cost of creating new games might decrease because it is no longer necessary to explicitly design every part of the gaming experience. In this thesis I am explicitly interested in the issue of PCG under a physics based gaming environment of which titles such as Angry Birds and Cut the Rope represent well known examples. Within the context of this work I concentrate on an ‘Angry Birds’ style of task. That is to say, the player has to use their skill in interpreting the Newtonian physics to aim Angry Bird projectiles with different properties to score points. Part of the process involves the destruction of objects that could be blocking the path to the target. Unlike previous work in PCG for this style of game I adopt an approach in which image data, possibly even captured by the user, represent the basis for designing the Angry Bird structure in which pigs are placed. The motivation for doing so is to increase the range of complexity of the resulting structures, while simultaneously reminding the reader of the original object. I believe that such an approach will both challenge and entertain the user.

The algorithm adopted first requires that the foreground object from the image is identified using image editing software, for which several applications are available. With the target object identified, a corresponding silhouette is extracted. The silhouette is used as the basis for defining a low resolution matrix expressing the basic location for square blocks. However, as soon as the original object (and therefore the corresponding matrix) has any overhang, then gravity will result in blocks free falling. I therefore also introduce a process for identifying the overhanging blocks and replace (sequences) of them with horizontal rectangles, such that the stability of the resulting structure is initially guaranteed. A final process provides for the automatic

placement of pigs within the structure, such that holes and supporting structures are introduced into the structure.

A database of 20 different source images is built in order to illustrate the relative generality of the approach. A wide range of source objects are considered, from the face of animals, to profiles of animals, to the profile of well known buildings, to ships and Jack-O-Lantern. Some builds are definitely more stable than others. However, the builds as a whole are generally much more complex than previously attempted by entries to the Angry Birds competition.

Future work will introduce a rule base to identify most likely sources of instabilities in the initial builds. Specific examples could be to include Newtonian physics into the assessment of rectangular blocks used to provide overhangs, and the introduction of minimal inter-block placing (stops blocks from ‘leaning’ against each other, where such leans appears to introduce instabilities).

Another potential area of development is to consider the use of some recent outcomes from deep learning in which objects are automatically identified in source images, therefore further removing the need for human intervention.

Appendix A

Level Screenshots

In this section, some sample images and Science Birds levels that are generated from these images are presented.

Figure A.1: Sample input image and output level: Cat

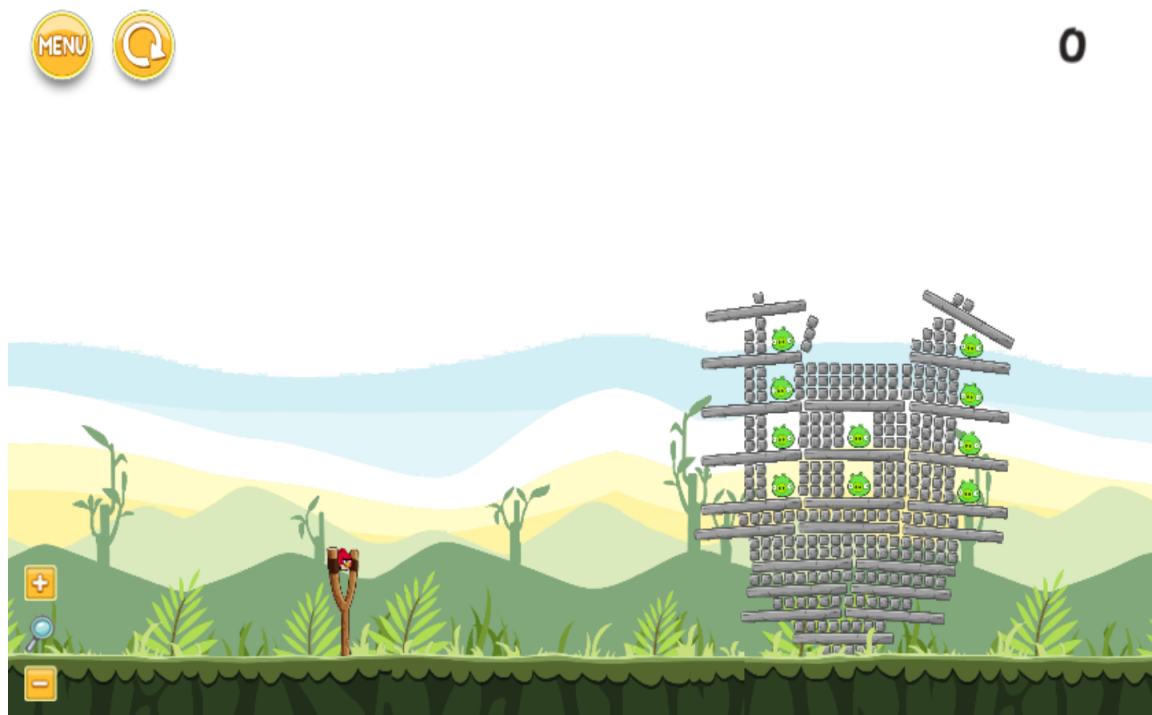


Figure A.2: Sample input image and output level: Cow



0

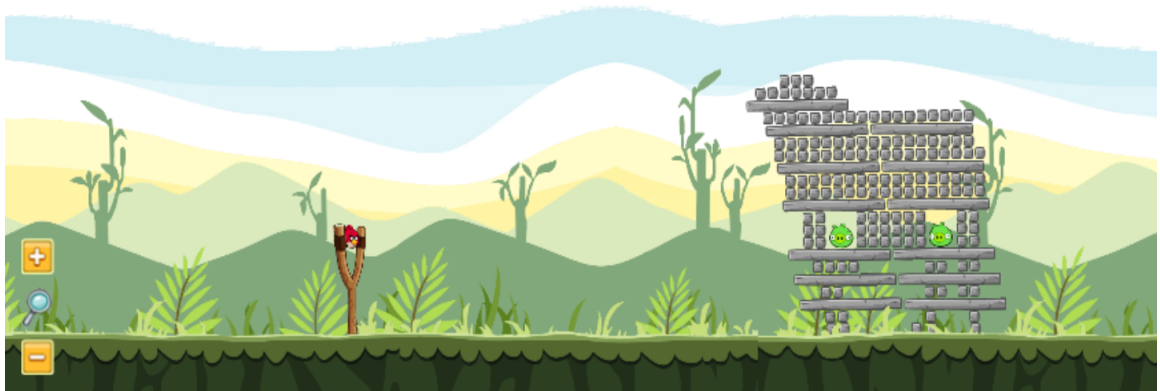


Figure A.3: Sample input image and output level: Dog



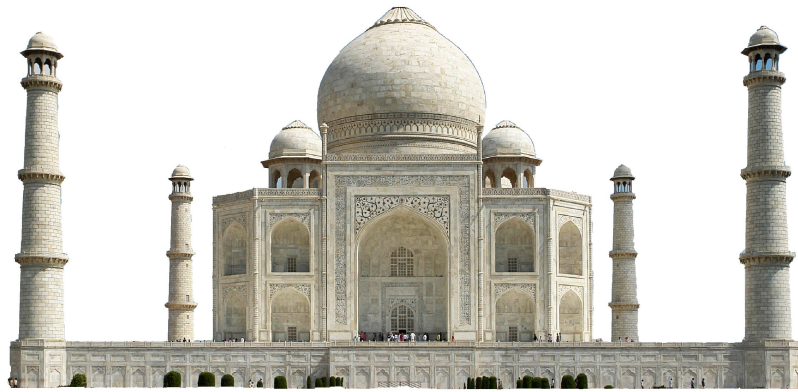
Figure A.4: Sample input image and output level: Hedgehog



0



Figure A.5: Sample input image and output level: Taj Mahal



0

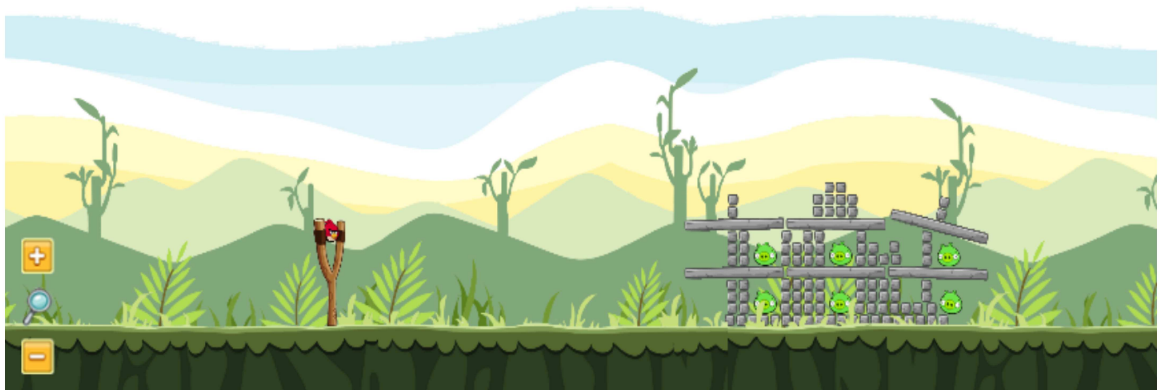


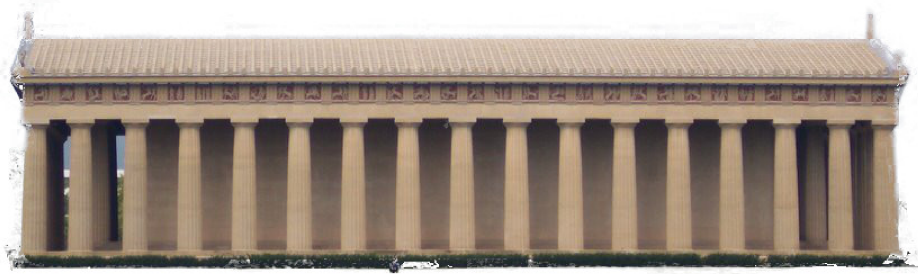
Figure A.6: Sample input image and output level: Triumphal Arch



0



Figure A.7: Sample input image and output level: Parthenon



0



Figure A.8: Sample input image and output level: Louvre Pyramid

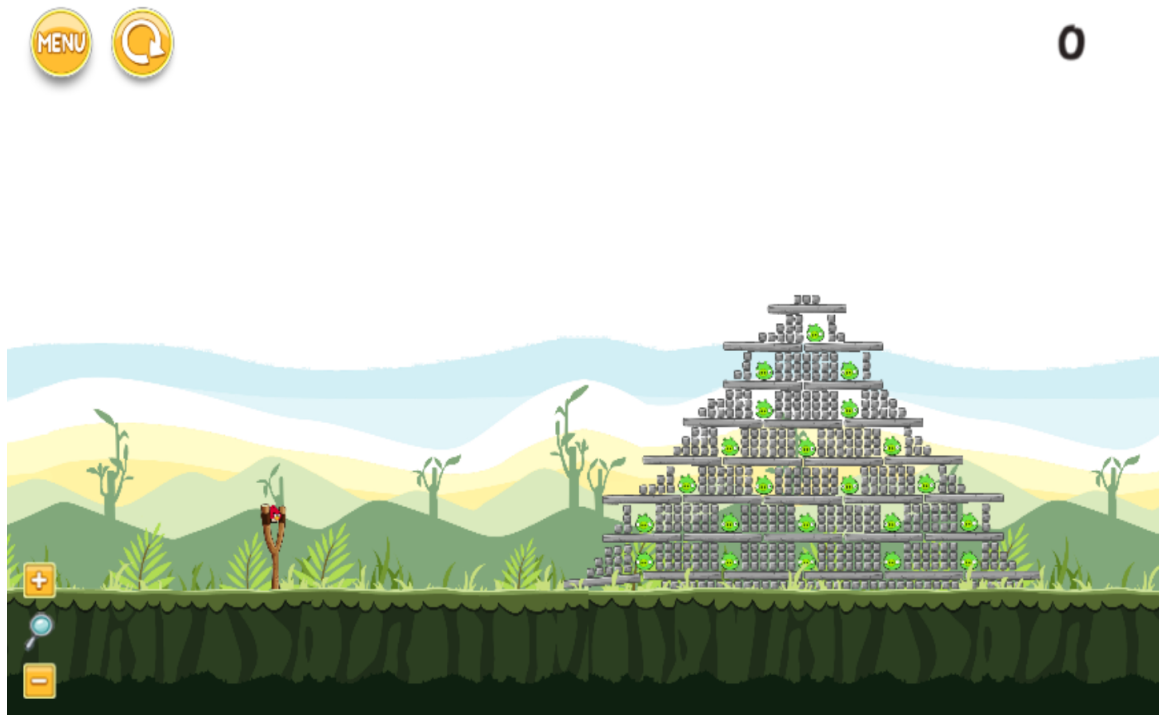


Figure A.9: Sample input image and output level: Grime

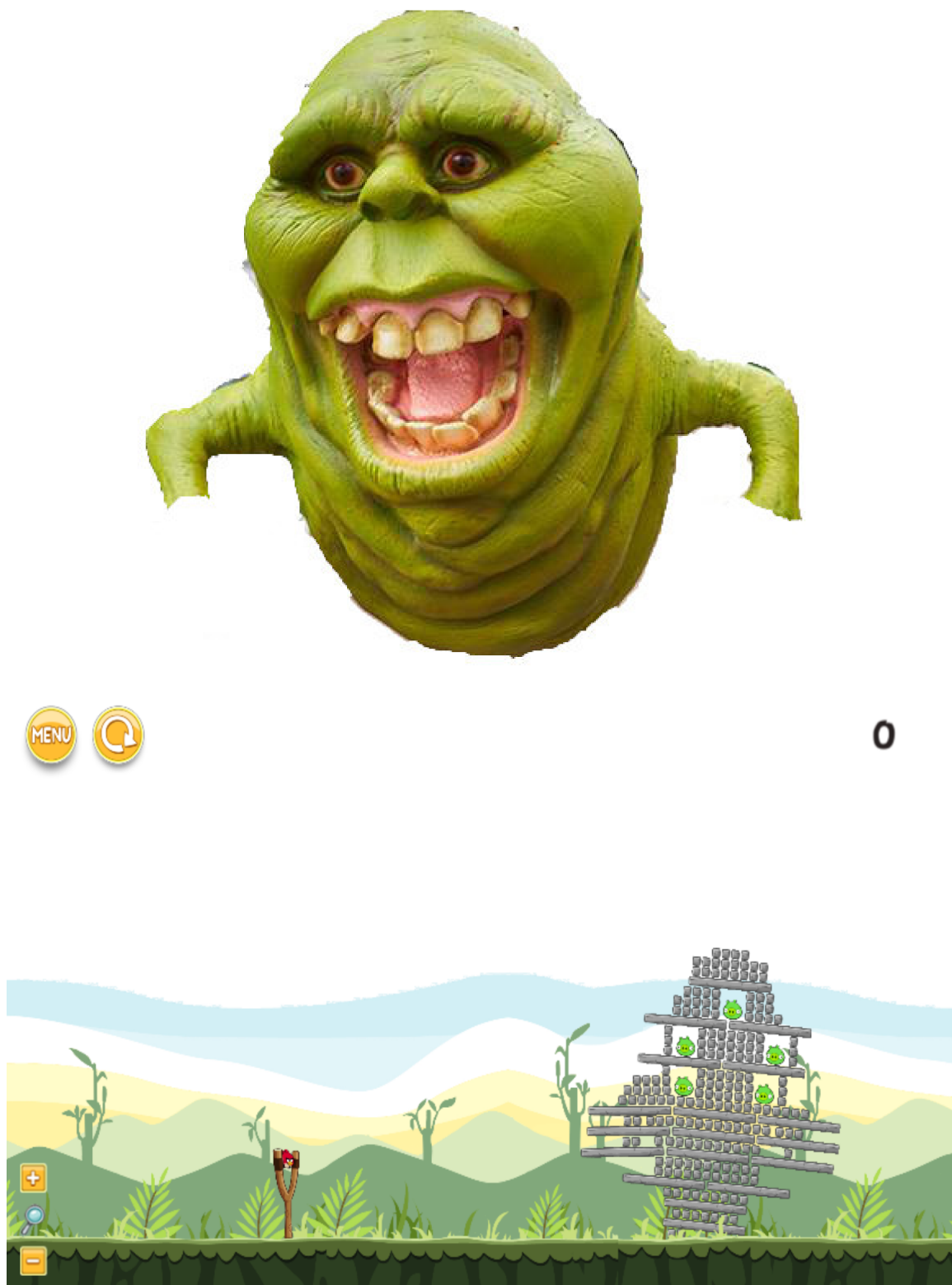


Figure A.10: Sample input image and output level: Jack-o-Lantern

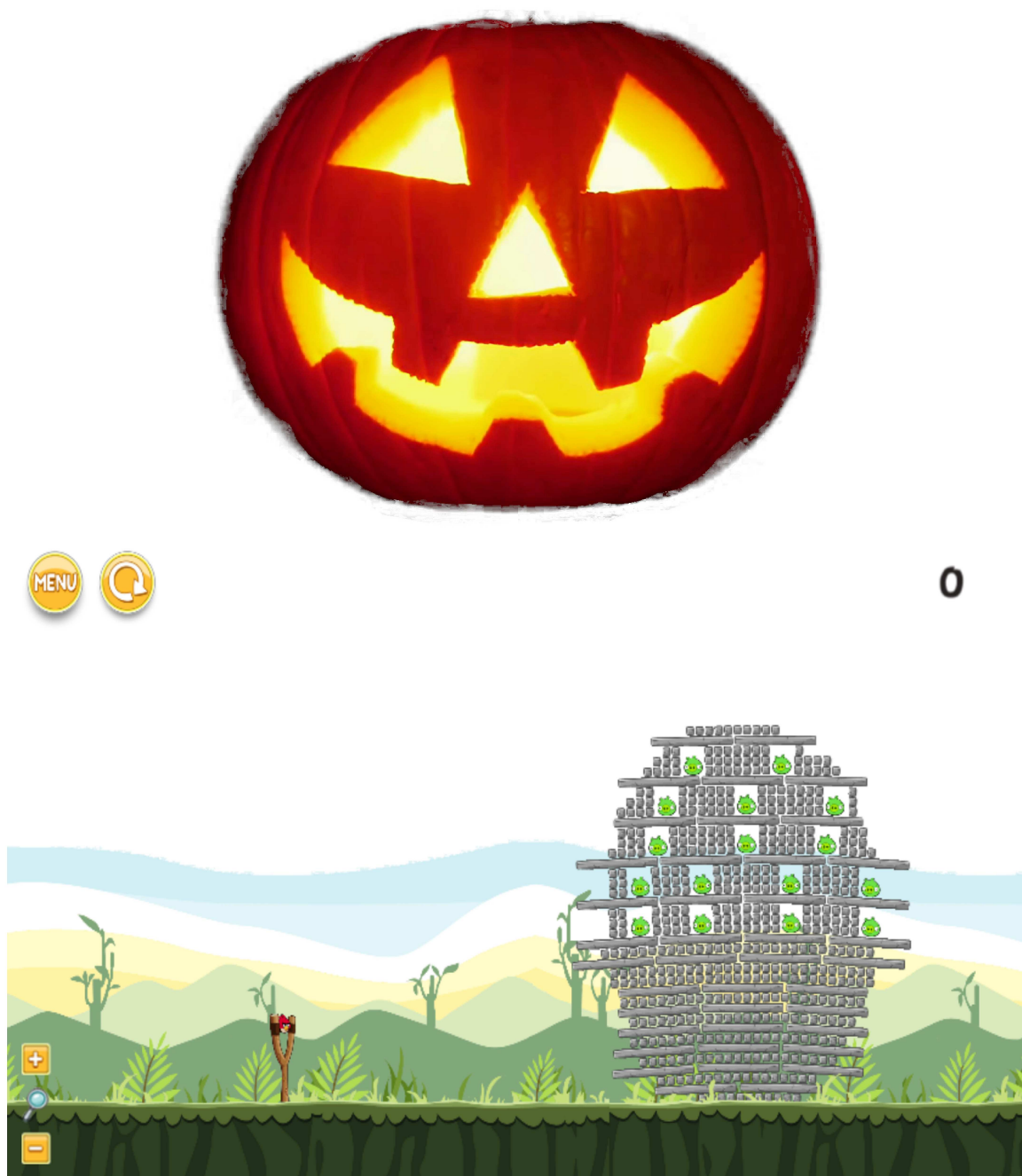


Figure A.11: Sample input image and output level: Haunted House



Figure A.12: Sample input image and output level: Candy Bowl

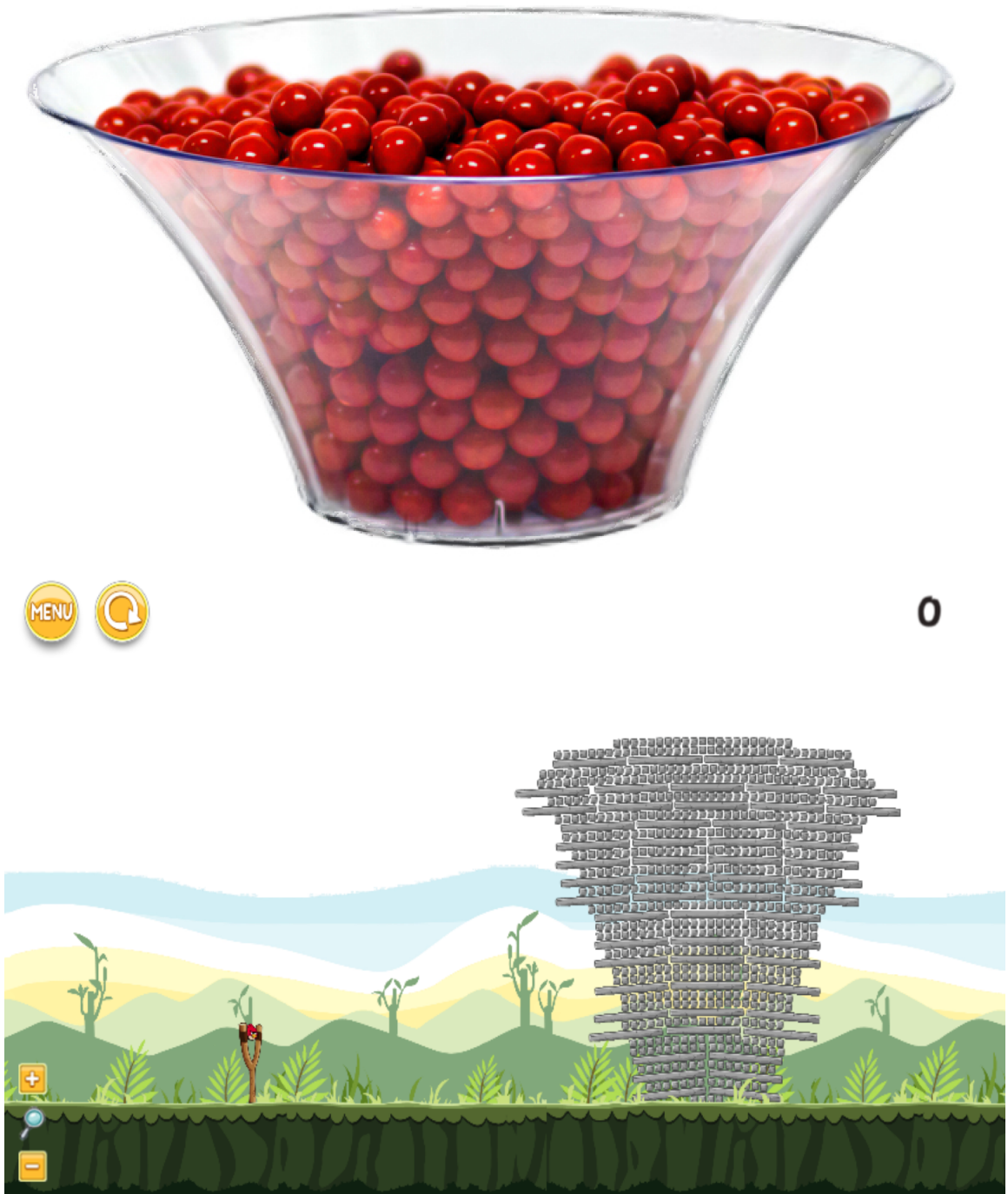


Figure A.13: Sample input image and output level: Chimneys

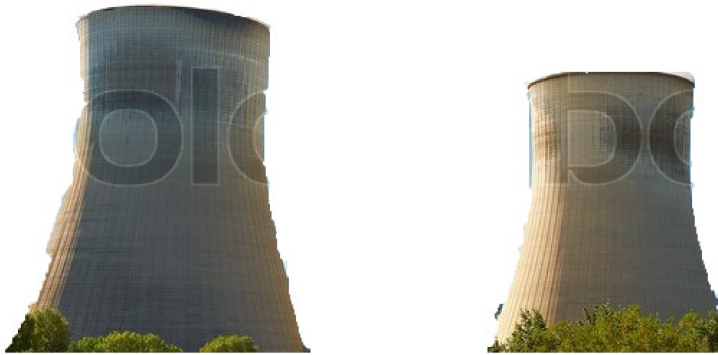


Figure A.14: Sample input image and output level: Water Wheel



0



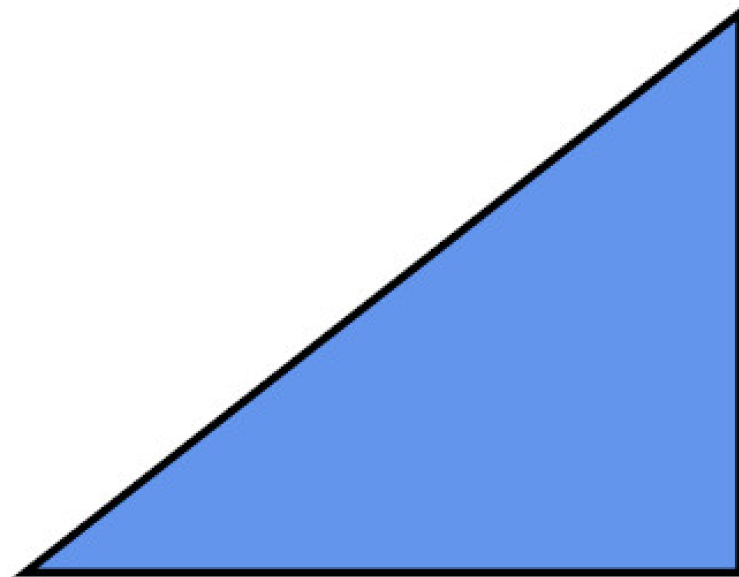
Figure A.15: Sample input image and output level: Combine Harvester



0



Figure A.16: Sample input image and output level: Inclined Plane



0

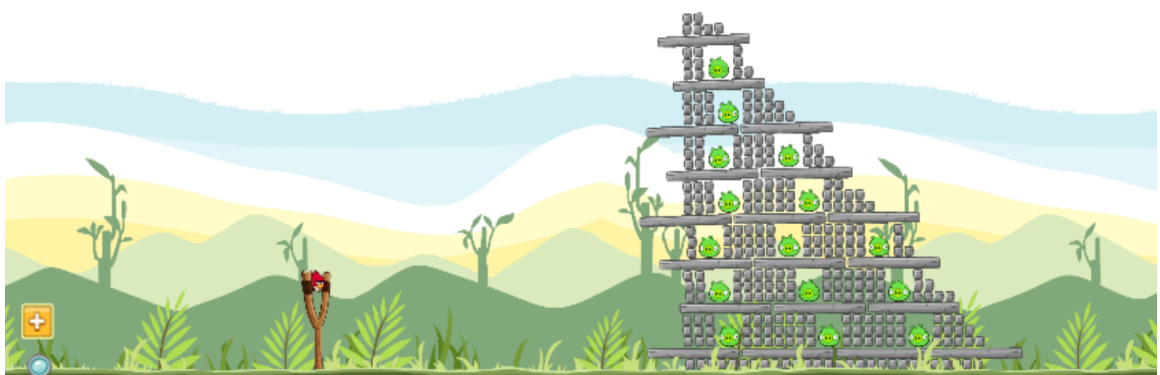


Figure A.17: Sample input image and output level: Catamaran

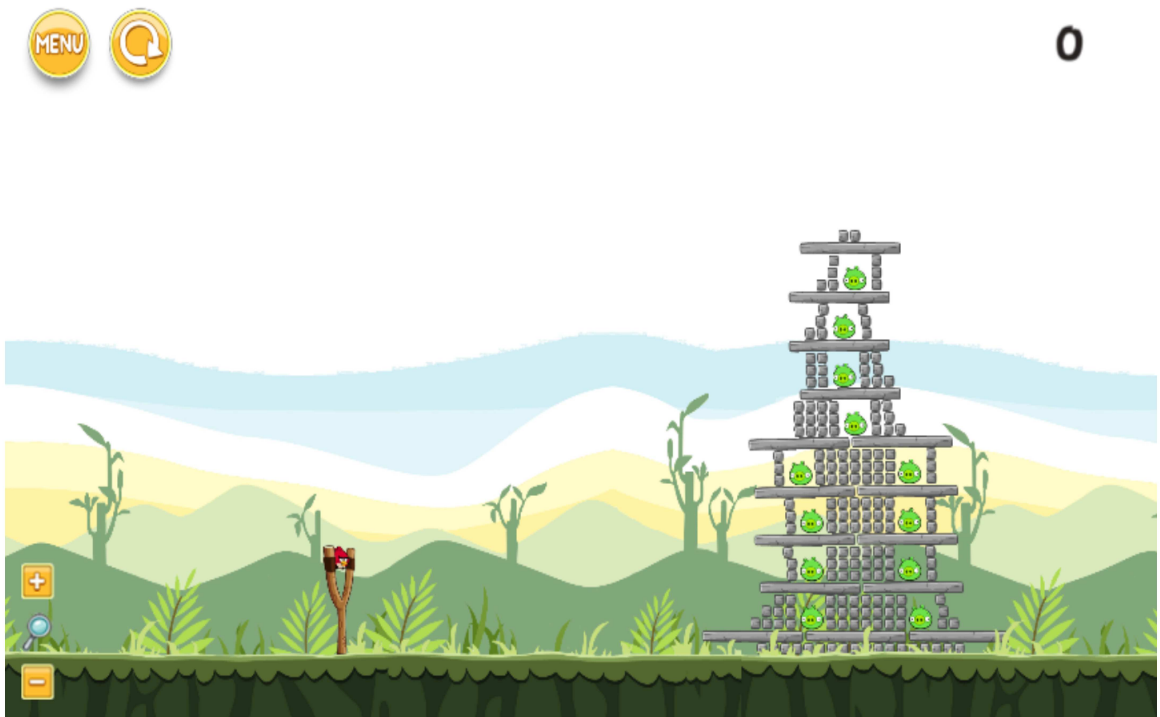


Figure A.18: Sample input image and output level: Ferry



0



Figure A.19: Sample input image and output level: Galleon



0

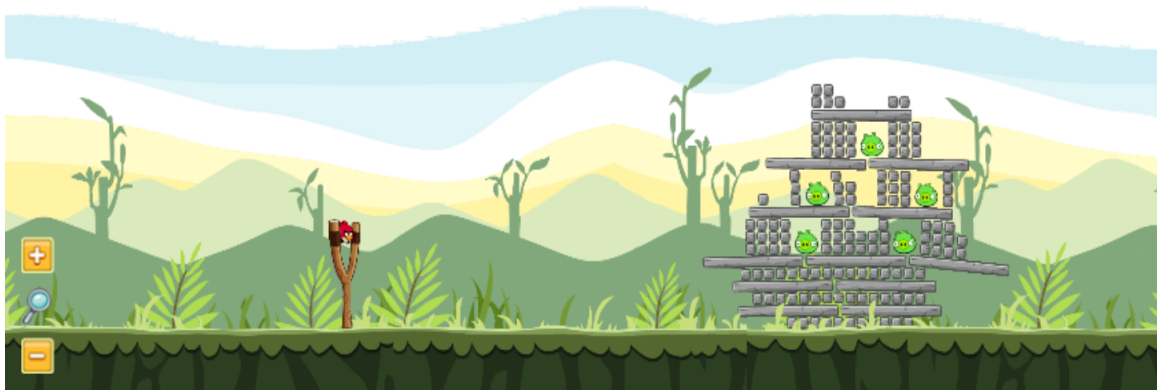
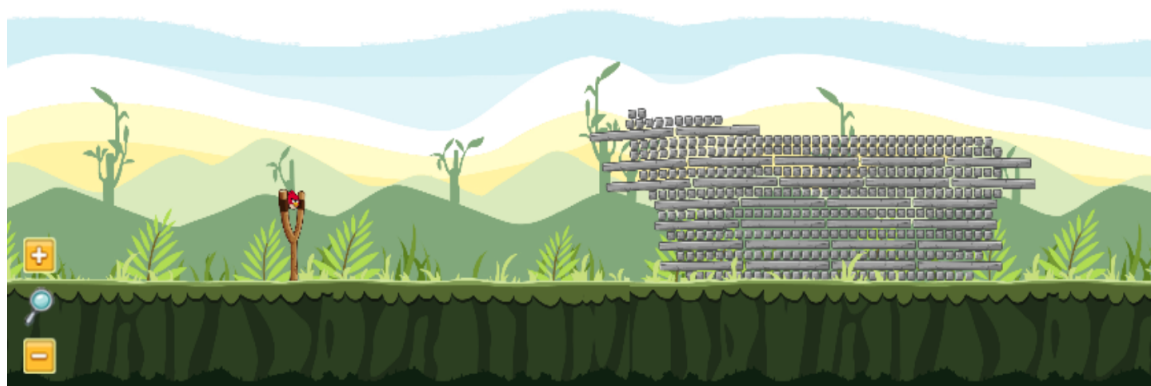


Figure A.20: Sample input image and output level: Rowboat



0



Appendix B

Parameter Values

These are some of the parameters present in the structure generator code [3].

| | | |
|---|------|-----------------------------------|
| Percentage of a mesh block needed to be covered for a primary block to be inserted | 50 | src/python/structure.py |
| Sigma value for the Gaussian filter | 10 | src/shell/generate-level.sh |
| Size value for the median filter | 10 | src/shell/generate-level.sh |
| Number of samples to sample the SVG 'path' element to get an SVG 'polygon' element from | 1024 | src/python/svg_path_to_polygon.py |

Table B.1: Hardcoded variables and their default values. These exist in the code not like a variable per se, but as a part of the code.

| | |
|------------------------------|---|
| LevelPath | Specifies the directory for the levels to be written under. |
| PrimaryBlock | Specifies the block to use as the main component of the structure. |
| PlatformBlock | Specifies the block to use to support the primary blocks without anything under. |
| NumberOfPrimaryBlocksOnXAxis | Specifies how many primary blocks should be inserted to cover the width of the structure. |

Table B.2: Configurable variables and their descriptions. These variables are able to be configured by the user through a configuration file.

Bibliography

- [1] Lucas Ferreira and Claudio Fabiano Motta Toledo. A search-based approach for generating angry birds levels. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8, 2014.
- [2] Lucas N. Ferreira and Claudio Fabiano Motta Toledo. Tanager: A generator of feasible and engaging levels for angry birds. *IEEE Transactions on Games*, 10(3):304–316, 2018.
- [3] Utku Gultopu. Science Birds PCG. <https://github.com/ugultopu/Science-Birds-PCG>, 2018.
- [4] Yuxuan Jiang, Tomohiro Harada, and Ruck Thawonmas. Procedural generation of angry birds fun levels using pattern-struct and preset-model. In *IEEE Conference on Computational Intelligence and Games, CIG 2017, New York, NY, USA, August 22-25, 2017*, pages 154–161, 2017.
- [5] Matthew Stephenson and Jochen Renz. Procedural generation of levels for angry birds style physics games. In *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2016, October 8-12, 2016, Burlingame, California, USA.*, pages 225–231, 2016.
- [6] Matthew Stephenson and Jochen Renz. Generating varied, stable and solvable levels for angry birds style physics games. In *IEEE Conference on Computational Intelligence and Games, CIG 2017, New York, NY, USA, August 22-25, 2017*, pages 288–295, 2017.
- [7] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):172–186, 2011.