A System for Procedurally Generating Puzzles for Games

by

Alec Thomson

S.B., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author . . . .
Department of Electrical Engineering and Computer Science
May 11, 2013

Certified by . . . .
Richard Eberhardt
Research Staff, MIT Game Lab
Thesis Supervisor

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A System for Procedurally Generating Puzzles for Games

by

Alec Thomson

Submitted to the Department of Electrical Engineering and Computer Science
on May 11, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

In this thesis, I designed and implemented a set of designer-targeted tools and libraries for procedurally generating puzzles for video games and interactive fiction. The system has the goals of producing solvable and flexible general-purpose puzzles through the use of simple tools targeted at small development teams. The system is implemented as two graphical tools built with the Java Swing Toolkit and a runtime library written for the popular Unity game engine. Two games have been built with earlier versions of the tools and one game was built with the most recent iteration of the tools to test the effectiveness of the system.

Thesis Supervisor: Richard Eberhardt
Title: Research Staff, MIT Game Lab

Acknowledgments

Foremost, I thank the MIT Game Lab and the former Singapore-MIT GAMBIT Game Lab for the inspiration and extensive support of this research.

Many thanks are due to Clara Fernandez-Vara for her inspiration, guidance, and support and Richard Eberhardt for his advice and insight as the project matured. Thanks go out to Michaela LaVan for her contributions to the original prototype and the teams behind *Symon* and *Stranded in Singapore* for essential inspiration and feedback. Special thanks go out to Andrew Grant for his constant high-quality advice and insight.

Labmates and colleagues have contributed to this work in countless ways: Philip Tan, Owen Macindoe, Abraham Stein, Sara Verrilli, Marleigh Norton, Generoso Fierro, Todd Harper, Jason Beene, Jason Begy, Mike Rapa, Matthew Weise, Naomi Hinchen, Kyle Miller, Raeez Lorgat, Daniel Vickery, Avril Kenney, and Alison Malouf.

Finally, I would also like to thank my friends and family for their continued unflinching support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While procedural content generation has been a common technique used in video games since their invention, most procedural content generation in commercial products and other research focuses on map generation, audiovisual generation, and environmental generation. In the few cases where procedural puzzle generation is explored, the focus is on very narrow forms of puzzles. The aim of my thesis was to research and develop much more general tools and ideas that can be used procedurally generate puzzles for a wide variety of games.

This kind of general procedural generation of puzzles represents an interesting technical challenge because of the requirements most games have for their puzzles. For a procedural puzzle generator to be useful, it must generate puzzles that are simultaneously solvable, intuitive, and engaging. For a puzzle generator to be adopted, it must be adaptable to a wide variety of games and genres. To solve this problem, the approach my thesis work has taken is to separate puzzles into discrete reusable units that can be recombined and customized by designers making use of simple editors. This approach culminated in the Puzzledice system outlined in chapter 2. The rest of this chapter outlines background work and defines the goals of the system in more detail.

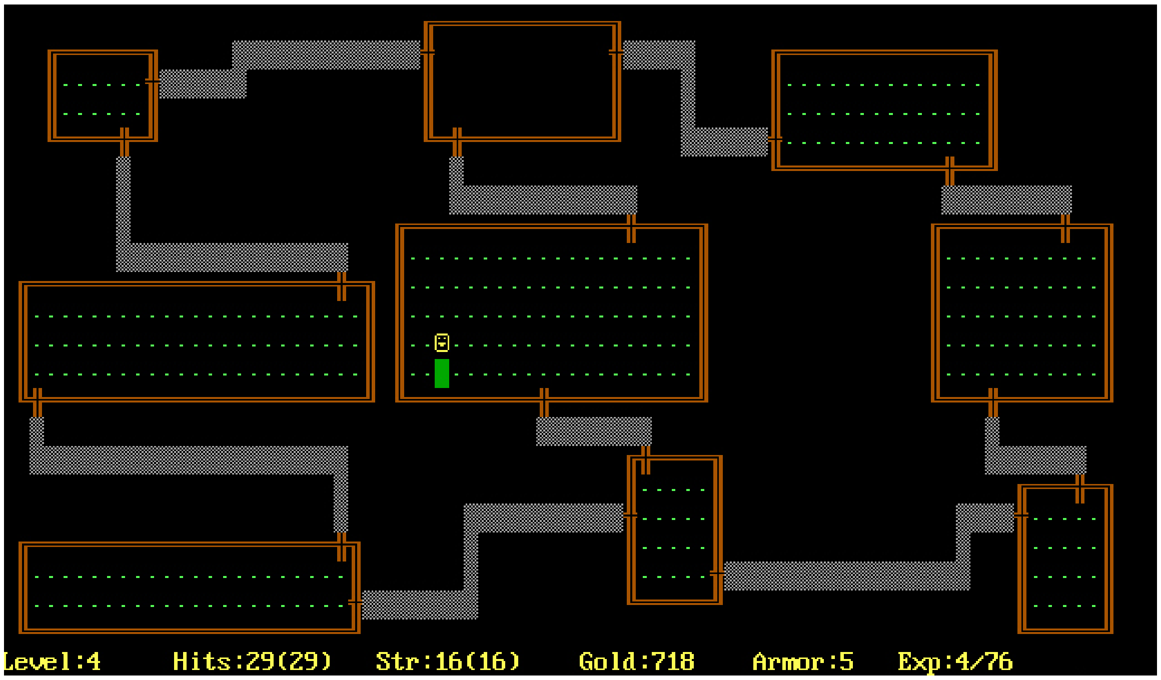


Figure 1-1: Screenshot of Rogue[15], one of the earliest examples of procedural generation in games. (source:[11])

1.1 Previous Work

There are a variety of benefits that can be gained from using procedural content generation as a component of video game development. First, procedural generation can be used for performance reasons. One notable example is the 1986 game *The Sentinel*, originally released for the BBC Micro, which used procedurally generated levels because the disk the game shipped on did not have enough space to contain manually designed levels.

Procedural generation can also be used to instill a game with a massive amount of re-playability. Games such as 1980's *Rogue*[15] and 2009's *Spelunky*[17] use procedurally generated levels to provide a unique experience every time the player plays.

Procedural generation can also be used to construct gigantic game spaces that it would be beyond the scope of a design team to manually create. Recent independent games *Minecraft*[10] and *Dwarf Fortress*[1] both use procedural generation to construct enormous worlds for players to explore.

Despite the obvious utility of procedural content generation for video games, not

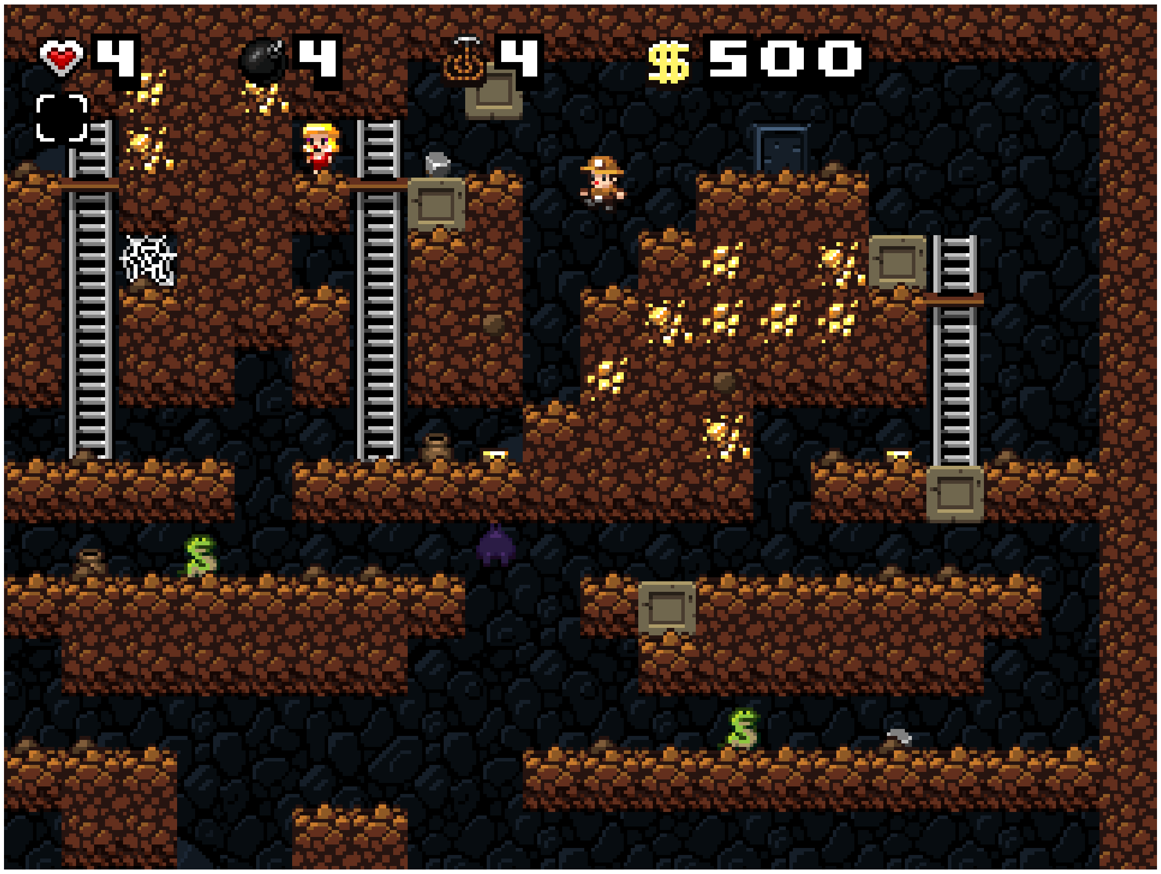


Figure 1-2: Screenshot of Spelunky[17], a modern example of procedural generation in games. (source:[12])

a lot of commercial products attempt to procedurally generate puzzles. Those that do tend to focus on a very narrow kind of puzzle (such as *Minesweeper*[9]) or allow procedurally generated puzzles to be unsolvable or optional (such as *The Binding of Isaac*[8]). The most likely reason for this is the fact that games tend to have very strict requirements for their puzzles. In general, designers want their puzzles to be *solvable* so the player is not frustrated by an unsolvable game, *intuitive* so the player can find a solution to the puzzle, and *engaging* so the player continues to play and enjoy the game. Procedurally generating puzzles that fulfill all of these requirements represents an interesting technical challenge.

There exists a lot of related ongoing and previous research into procedural content generation in games. While not specifically related to puzzles, a good amount of research has been done on procedural level generation for video games. Some examples include Compton and Mateas[3] and Dormans[5]. Specific to puzzle research, Ashmore[2] has researched procedural "lock and key" puzzles while Doran and Parberry[4] have constructed a procedural "Quest" model based on common structures of quests from MMORPGs.

Some of this research was ultimately the inspiration for *Symon*, a game produced during the GAMBIT summer program by a team led by GAMBIT Post-doc Clara Fernandez-Vara[6]. *Symon* was a point-and-click adventure game that attempted to procedurally generate traditional point-and-click adventure game "narrative" puzzles. The puzzles generated by *Symon* were very narrow in focus and ultimately consisted of a small subset of the kinds of narrative puzzles common to point-and-click adventure games and interactive fiction.

It was the benefits and drawbacks of the system used in *Symon* that directly inspired the development of the Puzzledice system which forms the basis of my research. An initial prototype of this system was developed by myself and a team of undergraduates at the Singapore-MIT GAMBIT Game Lab led by Clara Fernandez-Vara in the Spring of 2011. This early iteration of the system was integrated into the game *Stranded in Singapore* by a team of students during the 2011 GAMBIT summer session[7]. Direct feedback from the *Stranded* team was a large factor in the direction

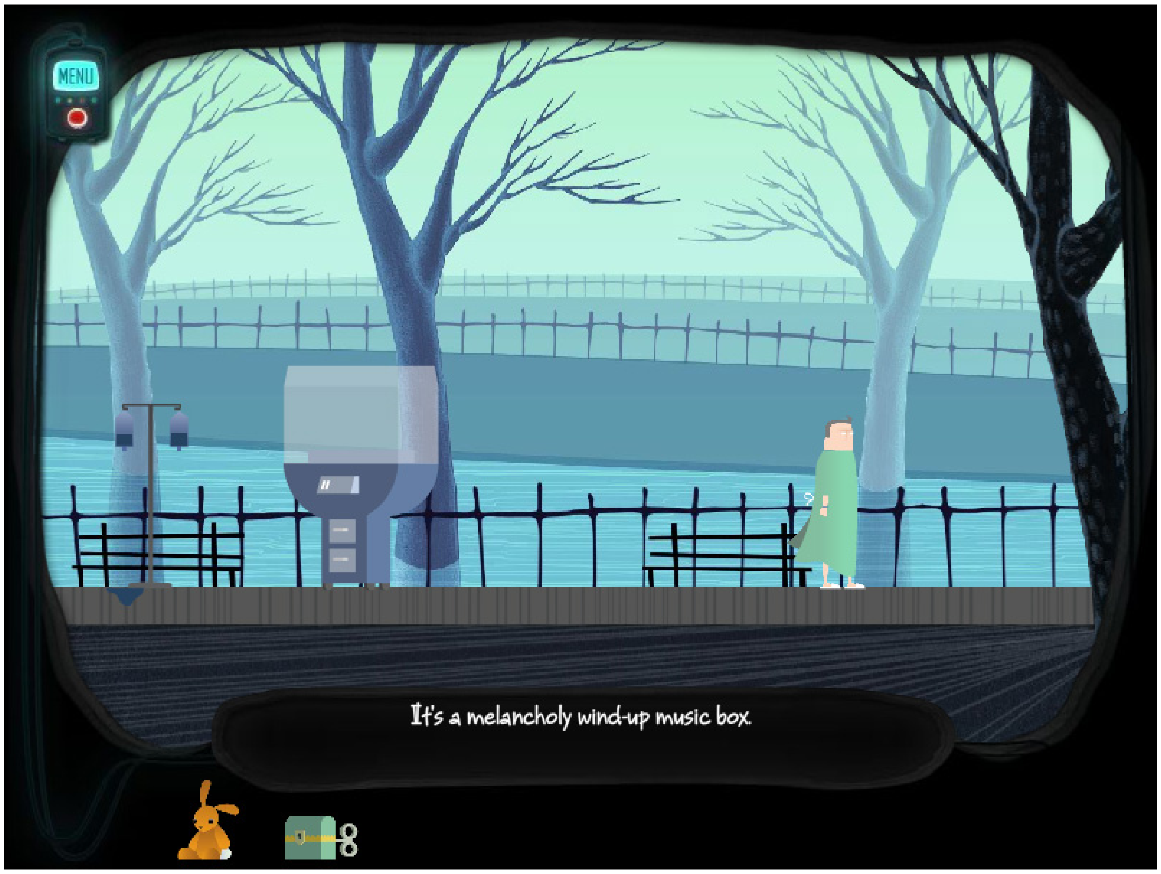


Figure 1-3: Screenshot of *Symon*[6], the primary inspiration for this research. (source:[14])

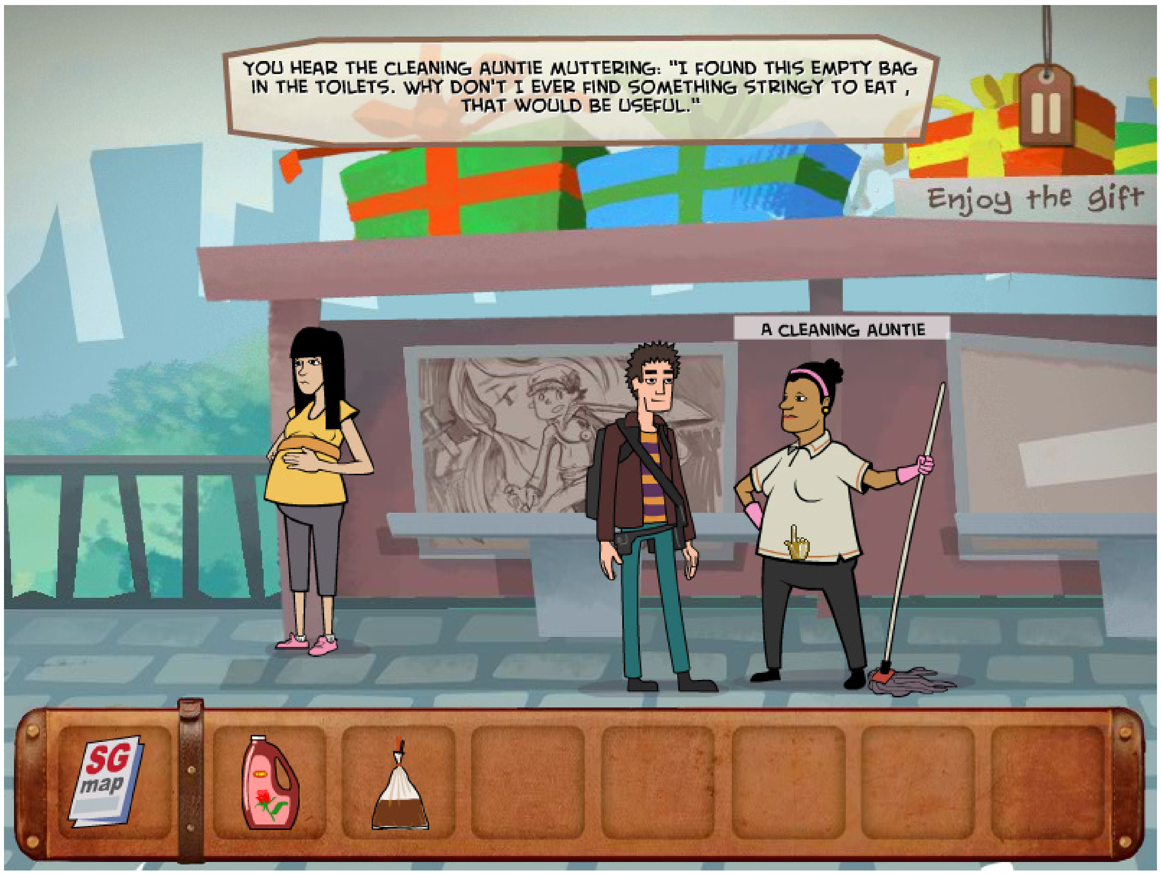

YOU HEAR THE CLEANING AUNTIE MUTTERING: "I FOUND THIS EMPTY BAG IN THE TOILETS. WHY DON'T I EVER FIND SOMETHING STRINGY TO EAT , THAT WOULD BE USEFUL."
Enjoy the gift
A CLEANING AUNTIE
SG map

Figure 1-4: Screenshot of *Stranded in Singapore*[7], a game made with an early iteration of the system. (source:[13])

of my research. Since the summer of 2011, my contributions to the system outlined in this thesis include managing a complete overhaul of the puzzle generation library, porting the puzzle generation library to the Unity game engine, designing and implementing the latest version of two graphical editors that interface with the library, and constructing a sample game (outlined in Chapter 6) to evaluate the system.

1.2 Problem Description and Goals

The ultimate goal of my research was to develop a designer-targeted system for procedurally generating puzzles that could be used in a variety of games. To meet these requirements, three overarching goals were defined for the system. These goals are referred to as solvability, generality, and usability. Each of these goals is described in more detail below.

1.2.1 Solvability

One of the major challenges of a procedural puzzle system for games is to produce puzzles that are guaranteed to be solvable by a player. A system that produces unsolvable puzzles is unlikely to be useful for designers or adopted in existing genres of games. While there exist many games that relax the solvability requirement ([17], [8]) to reduce complexity, relaxing solvability requires a good understand of the specific game being implemented and can lead to unexpected results. Therefore, solvability is necessary to achieve generality and usability. Since solvability is a difficult goal to achieve and inspires much of the design decisions of the system, it is listed as a major goal on its own.

Additionally, the original implementation of *Symon*[6]–the inspiration for this research–produced puzzles that could be placed into an unsolvable state, a problem which ultimately hurt the game and required a patch to fix after the game was released. With this case in mind, solvability was a major goal for my system.

1.2.2 Generality

For the system to be useful for designers and game developers, it must be possible to integrate the system with a variety of games in a variety of genres and platforms. The system should also be highly extensible so that developers can customize the system to the individual needs of their games. These are the requirements of generality.

As will be outlined in the following chapters, my system achieves generality by presenting a small set of puzzle primitives (building blocks) that can be recombined and extended for different purposes. Additionally, the system provides a highly extensible database that allows developers to customize and extend the output of the puzzle generator depending on how they integrate the system with their games. Chapter 6 presents a sample implementation game and describes how the system could be integrated with a large variety of games.

1.2.3 Usability

For the system to actually be useful to game developers, it must be usable enough to not produce undue amounts of frustration on the part of the developers. This is the requirement of usability.

To provide usability, my system is designed with a team of technical and non-technical developers in mind and provides tools for both. While a technical developer is necessary to integrate my system with a game, I also provide two graphical tools that were designed for use by non-technical designers to add content to a game integrated with my system. Additionally, I provide a runtime library written for the Unity game engine, a popular engine that is widely used by technical and non-technical game developers alike.

1.3 Roadmap

The system I present in the rest of this thesis is known as the "Puzzledice" system and is composed of several parts. The high level design of this system is outlined in chapter 2 while individual components of the system are described chapter 3, chapter 4, and chapter5. A sample game that integrates with the system is described and analyzed in chapter 6, while chapter 7 concludes with an eye towards limitations and future work.

Ultimately, I believe this system is novel because it is the first attempt at a general purpose procedural puzzle system that satisfies the three goals of solvability, generality, and usability. My hope is that this system can be used as a starting point for developers and researchers who wish to study or apply procedurally generated puzzles to their games.

Chapter 2

Puzzledice System Design

2.1 Overview

The high-level design of the Puzzledice system is split up into three major components, a puzzle generation library that's implemented as a runtime library for the Unity game engine, a graphical tool for editing a database of puzzle items and relationships, and a graphical tool for building puzzle map structures that the generator requires as input. Each of these components is briefly described below, with a more detailed description in the following chapters.

2.2 Puzzle Generator

The puzzle generator, described in greater detail in Chapter 3, is a runtime library implemented for the popular Unity game engine. The purpose of the puzzle generator is to take designed input (from the graphical tools) and produce object manipulation puzzles in abstract form as output.

More concretely, the puzzle generator takes as input a database of items and their relationships (created in the database editor tool) along with a puzzle map defining the desired interaction flow of the output puzzle. The puzzle generator then returns a list of abstract items and relationships that the integrating game can spawn to implement the puzzle.

The puzzle generator uses a basic approximation algorithm to fit database items and relationships to a set of constraints defined by the puzzle map. As proven in Appendix A, the problem solved by the puzzle generator is NP-Complete, justifying the use of an approximation algorithm despite a negative effect on usability.

2.3 Database Editor

The database editor, described in greater detail in Chapter 4, is a graphical tool used to define the items and relationships that the puzzle generator can use when generating puzzles. It was designed with non-programmers in mind as the target users.

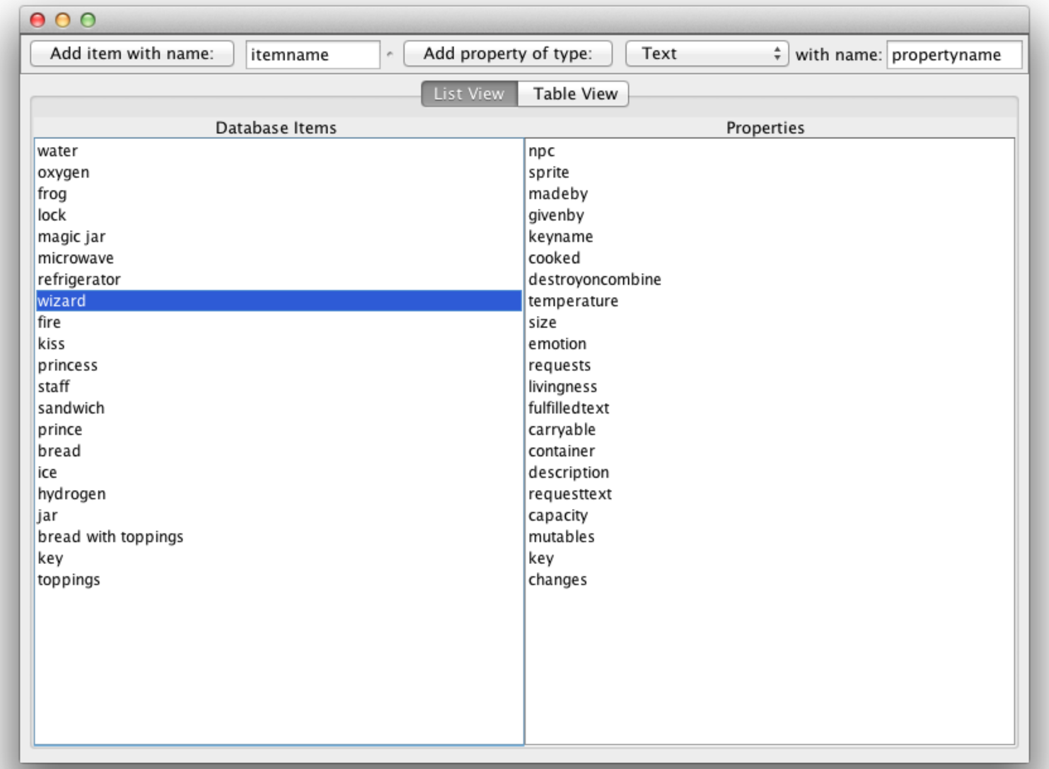
The tool is a basic graphical editor implemented with the Java Swing Toolkit that includes multiple data views and a lot of customizability. The editor exports to XML for easy portability.

The database editor is used primarily to produce content for the puzzle generator to use when generating puzzles. The database editor features many customizable systems, including custom properties and programmatic database extensions. The database editor is general enough that it can be useful as a design tool for games unrelated to the Puzzledice system.

2.4 Puzzle Map Editor

The puzzle map editor, described in greater detail in Chapter 5, is a graphical tool used to build the puzzle maps that the puzzle generator uses to determine the abstract "shape" of the puzzle it produces. This "shape" represents the flow of interactions the player engages in to solve the puzzle and is described in more detail in Chapter 3. The puzzle map editor was designed with non-programmers in mind as the target users.

The tool is a simple graphical editor implemented with the Java Swing Toolkit that includes a graph visualization of the puzzle map. The editor only allows acyclic



itemname

Text

propertyname

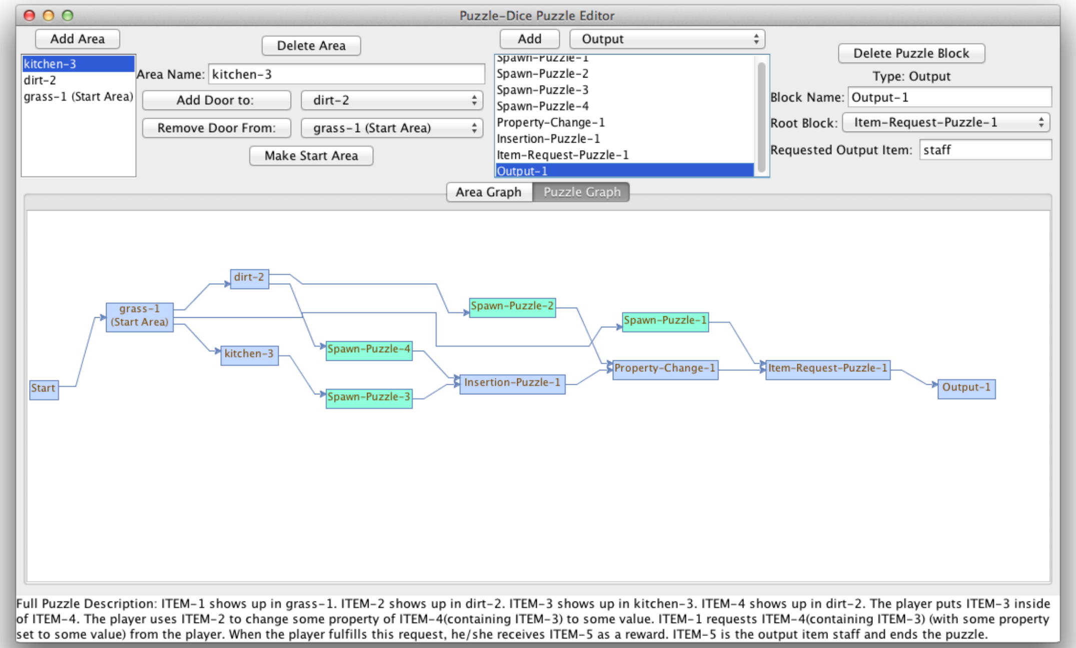
Database Items

water
oxygen
frog
lock
magic jar
microwave
refrigerator
wizard
fire
kiss
princess
staff
sandwich
prince
bread
ice
hydrogen
jar
bread with toppings
key
toppings

Properties

npc
sprite
madeby
givenby
keyname
cooked
destroyoncombine
temperature
size
emotion
requests
livingness
fulfilledtext
carryable
container
description
requesttext
capacity
mutables
key
changes

Figure 2-1: Screenshot of the Database Editor



Area Name: kitchen-3

Add Door to: dirt-2

Remove Door From: grass-1 (Start Area)

Type: Output

Block Name: Output-1

Root Block: Item-Request-Puzzle-1

Requested Output Item: staff

Full Puzzle Description: ITEM-1 shows up in grass-1. ITEM-2 shows up in dirt-2. ITEM-3 shows up in kitchen-3. ITEM-4 shows up in dirt-2. The player puts ITEM-3 inside of ITEM-4. The player uses ITEM-2 to change some property of ITEM-4(containing ITEM-3) to some value. ITEM-1 requests ITEM-4(containing ITEM-3) (with some property set to some value) from the player. When the player fulfills this request, he/she receives ITEM-5 as a reward. ITEM-5 is the output item staff and ends the puzzle.

Figure 2-2: Screenshot of the Puzzle Map Editor

puzzle maps to be created in keeping with the requirements of the puzzle generator. The editor exports to XML for easy portability.

Puzzle maps produced by the puzzle map editor can be used by a designer to exert greater control over the types of puzzles produced by the generator and provide enough features for a designer to completely remove non-determinism from the puzzle generator.

2.5 Motivation for Design

The design of the system was motivated by the goals outlined in Chapter 1 and the experiences gained from the system implemented in *Symon*[6]. Briefly:

- The puzzle generator was based on the generator used in *Symon*, with an emphasis on greater usability, solvability, and generality. In particular, I wanted the new system to be able to generate puzzles that the system used in *Symon* was unable to generate, the "Container" puzzle being the canonical example.

Additionally, the system in *Symon* produced puzzles that could be placed in an unsolvable state by the player, so the generator presented in Chapter 3 was designed heavily with solvability in mind.

- The database editor was developed in response to the difficulties *Symon*'s designer had crafting inputs to *Symon*'s puzzle generator. It has gone through a few iterations before arriving at the design and implementation presented in Chapter 4. The design was built with usability and generality in mind.
- The Puzzle Map Editor was developed in response to feedback from the team that developed *Stranded In Singapore* using an early iteration of these tools[7]. The team found that a tool for building puzzle maps was necessary for the development of their game and prototyped a simple version. The final tool presented in Chapter 5 was based on this simple version and designed primarily with usability in mind.

Chapter 3

Puzzledice Puzzlegen Algorithm

This section expands on the previous chapter's short description of the puzzle generator. It describes the inputs to the generator, the abstract outputs produced by the generator and some advice on how these inputs and outputs can be integrated with a game in a general way. The puzzle generator algorithm (referred to as *puzzlegen*) relies on the behavior of many puzzle "building blocks" that correspond to discrete areas or puzzle relationships. These building blocks all follow a very similar process that is described in Section 3.4. A detailed description of all the building blocks built for the system is included in Appendix B.

To illustrate how these different inputs and outputs are used, the following scenario of a simple "Combine" puzzle (illustrated in Figure 3-1) is presented as an example: The game presents the player with two objects, a "Hydrogen" item and an "Oxygen" item. The player combines these two items to produce the desired "Water" item.

3.1 Inputs to the Puzzle Generator

There are two main inputs to the *puzzlegen* algorithm: a database of items that can exist in the world and a recursively defined puzzle map representing the abstract structure of the puzzles to be generated. Each of these components is described in greater detail below and in the following chapters.

Additionally, the *puzzlegen* algorithm takes as input the name of an item to gener-

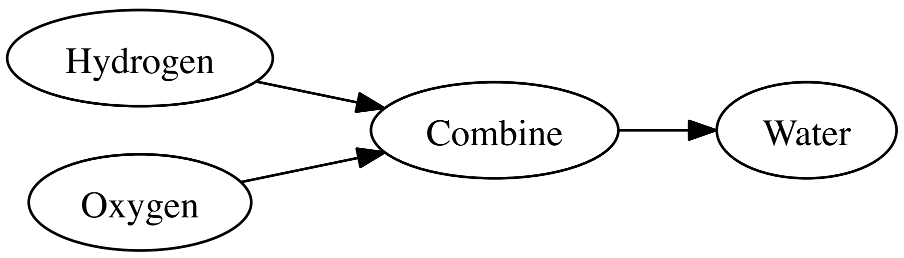


Figure 3-1: An example of a simple "Combine" puzzle

ate as the "reward" for solving the entire puzzle structure. For an integrating game, this reward is an item defined in the database that can be produced as the result of game interactions. For example, the reward item might be a physical item the player seeks to acquire (such as "The Amulet of Yendor") or it might be an abstract item recognized by the integrating game as a signal that the game is now complete (such as "Game-Complete-Trigger-22"). The puzzle generator requires this reward input primarily to support the elegant recursive structure of the puzzle map and for use as a starting point when generating a complete puzzle.

3.1.1 Item Database

The item database is a collection of items that can exist in the game world along with their properties and relationships to other items in the database. An item database can be manually constructed in code, procedurally generated by use of "Database Extensions" (see 4.1.3), or built with the graphical database editor tool described in Chapter 4.

The puzzle generator uses the item database to select items for use in its puzzles. It also uses the relationships between items in the database to determine which items are appropriate for different pieces of the generated puzzle.

While generating the scenario in Figure 3-1, the *puzzlegen* algorithm will first look at the "Water" entry in the item database. It will then determine that water can be constructed by combining the “Hydrogen” and “Oxygen” items. The *puzzlegen* algorithm then uses this information to construct the described scenario.

3.1.2 Puzzle Map

The puzzle map represents the structure of the puzzle we hope to generate. It can be viewed as a map of all the actions a player takes while moving from the beginning of the puzzle to the end of the puzzle. The puzzle map is composed of a set of “puzzles” and “areas” and the connections between them. These components of a puzzle map are collectively referred to as “building blocks” in this writeup and are described in more detail in Appendix B.

Puzzle building blocks are objects that represent a single set of actions the player performs to get from a set of input items (hydrogen and oxygen in the Figure 3-1), to an output item (water in the Figure 3-1 scenario).

Area building blocks are objects that represent a set of “rooms” within the game world that are accesible from each other. This notion of accessible space is important for the puzzle generator because it assumes the integrating game will involve a player character moving between spaces while interacting with items. The accessibility of certain spaces is therefore essential to solvability as the player must be able to access items to interact with them. The original version of the puzzle generator had no notion of where items were to be placed in relation to each other, but this feature became necessary to implement “Door Unlock” puzzles that relied on accesible space.

A puzzle map can be manually constructed in code or built with the graphical puzzle map editor tool described in Chapter 5. Figure 3-2 presents a sample puzzle map that represents the scenario in Figure 3-1. Each of the available building blocks is described extensively in Appendix B.

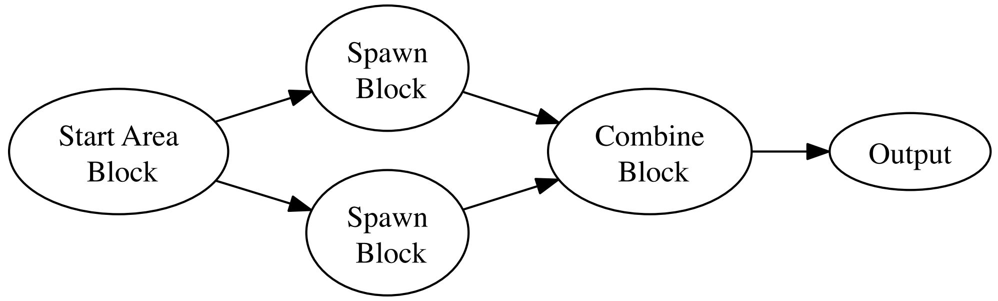


Figure 3-2: An example puzzle map for the scenario in Figure 3-1

3.2 Outputs of the Puzzle Generator

The output of the *puzzlegen* algorithm is a list of items and relationships to spawn in the world. Below is a more detailed description of these outputs.

3.2.1 Puzzle Relationships

The list of relationships produced by the *puzzlegen* algorithm represents a list of abstract relationships necessary to produce the puzzles created by the generator. Relationship is a rather loose term that can describe a number of different things. Here are a few examples of relationships used by existing building blocks of the system:

- A **Combine Relationship** represents a relationship between two "ingredient" items that can be combined to produce a third item. The relationship contains the ids of the two ingredient items along with a copy of the result item (and all properties it possesses).

- A **Property Change Relationship** represents a relationship between two items where one item can be used to change a property of the other. The relationship contains the names of the "changer" and the "changee" along with the name and desired value of the property to change.

- An **Insertion Relationship** represents a relationship between two items where one item can be inserted into the other.

- An **Item Request Relationship** represents a relationship where one item (possibly an NPC) "requests" another item from the player and offers the player a "reward" in return.

- An **Area Connection Relationship** represents a required connection between two areas. While additional "auxiliary" connections might be made between areas, an area connection relationship outlines a connection that *must* exist for the puzzle to be solvable. The relationship contains the unique IDs of the two areas requiring the connection.

Developers of integrating games are given freedom to decide how to implement the relationships listed by the puzzle generator. In the case of the Figure 3-1 scenario, the only relationship generated would be a combine relationship between the "Hydrogen" and "Oxygen" items to produce the "Water" item.

3.2.2 Puzzle Items

The list of items produced by the *puzzlegen* algorithm represents a list of items to be spawned in the game world. Each item includes a list of properties for the item to possess. These properties can include properties relevant to the generated puzzles and properties relevant to the integrating game. The properties are primarily copied from properties contained in the database entry for a given item, with a few exceptions in the case of property change puzzles.

Among the properties the items has is a unique ID for the area in which an item should be spawned. This information is mainly required for "Door Unlock" puzzles (as the items required to unlock a door must be accessible without opening the door for the puzzle to be solvable).

As is the case with relationships, developers of integrating games are given freedom to decide how to implement the items listed by the puzzle generator. To achieve

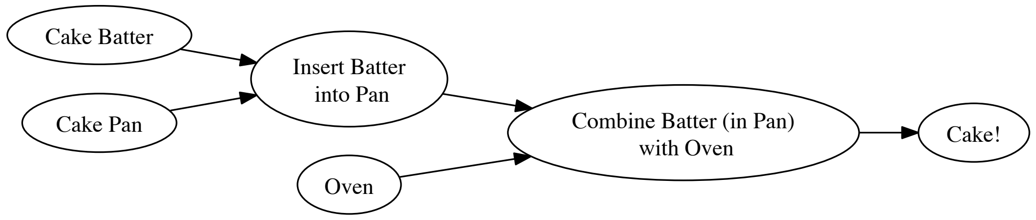


Figure 3-3: One process for building a cake with item relationships

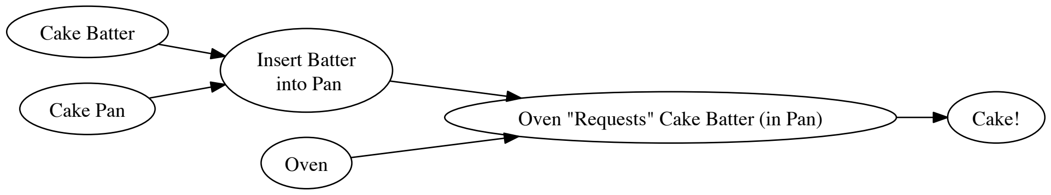


Figure 3-4: Another process for building a cake with item relationships

generality, I attempted to provide a general enough description of “item” for this information to be useful for a wide variety of games.

In the case of the Figure 3-1 scenario, the list of items to generate would include the “Hydrogen” and “Oxygen” items with both being spawned in the player’s starting area.

3.3 Generality of Puzzle Generator Output

While the outputs outlined above may seem limiting, they actually account for many situations, some of them unclear from the abstract relationships. For example, consider an in-game situation where the player needs to bake a cake.

Intuitively, the player would make a cake by making the batter and sticking it in the oven. This process can be described a variety of ways with the puzzle relationships. The relationship between the cake batter and the oven could be a combine relationship where some database property indicates that the oven should not be destroyed on the

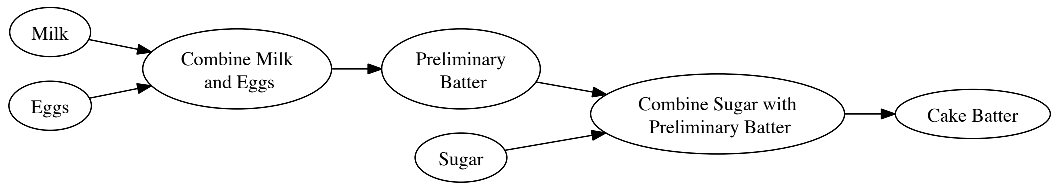


Figure 3-5: Process for building cake batter with combine relationships

combine, (such as in Figure 3-3). Alternatively, the relationship could be an item request relationship where the oven "requests" the cake batter and offers the baked cake as a "reward" (such as in Figure 3-4). Whatever the relationship between the batter and the oven, it is unlikely that the batter will be portable by the player. Therefore, an insertion relationship between the batter and the cake pan will also be necessary. The batter itself is constructed by mixing ingredients together, a process that might be represented by a cascade of combine relationships such as in Figure 3-5.

This example illustrates how the relationships produced by the generator can correspond to a wide variety of situations and puzzles within a game, meeting the goal of generality.

3.4 General Pattern of Puzzle Generation

The actual details of most of the *puzzlegen* algorithm depend on the building blocks used to construct the puzzle map. Fortunately, almost all of the building blocks follow a very general pattern when generating puzzles. An individual building block is only aware of its direct input building blocks and the requested output it received from its output block, as shown in Figure 3-6 and thus can operate independently of other blocks in the puzzle map. This section describes the inner workings of the *puzzlegen* algorithm in more detail.

When a building block is asked to generate a puzzle, it is given the name of a desired output item along with a list of desired properties for the output item to have. With this information, the building block then follows a sequence of steps

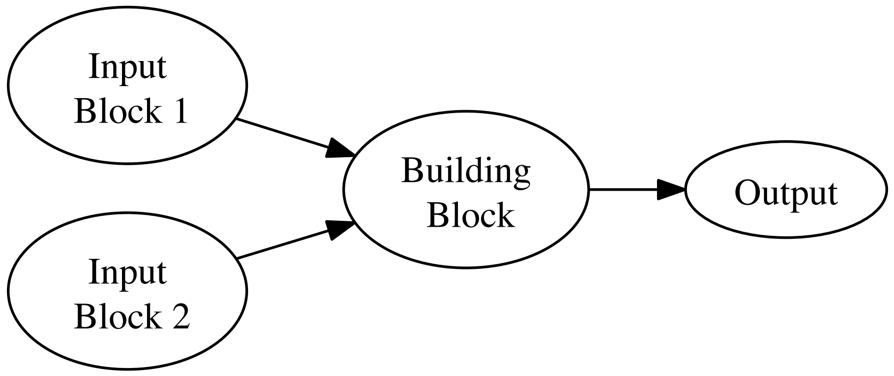


Figure 3-6: General structure of a building block

```
public bool generateOutput(String itemName) {
  if (Database.itemFullySpawned(itemName))
    return false;
  PuzzleItem item = Database.spawnItem(itemName);
  applyPropertiesToSpawnedItem(item, this.desiredProperties);
  this.outputItem = item;
  return true;
}
```

Figure 3-7: Pseudocode for simplified version of the output generation step

outlined below. If the building block determines it can't continue during any of these steps, it returns a failure to the caller that asked it to generate a puzzle (either another building block or the initial call to the *puzzlegen* algorithm). As described below, when inputs to a building block return a failure, the building block simply tries another combination until it runs out of options, at which point it returns a failure.

Step 1: Attempt to generate the output item

Summarized in Figure 3-7, the building block first attempts to generate the output item requested of it with the desired properties. If for some reason the output item

```
public bool generateInputs() {
  List<StringPair> possibleInputItemNames =
  findPossibleInputItemNames(this.outputItem);
  randomShuffle(possibleInputItemNames);
  for (StringPair itemNamePair : possibleInputItemNames) {
    String itemName1 = itemNamePair.String1;
    String itemName2 = itemNamePair.String2;
    if (!areCarryable(itemName1, itemName2))
      continue;
    PuzzleOutput output1 =
    this.input1.generatePuzzle(itemName1);
    PuzzleOutput output2 =
    this.input2.generatePuzzle(itemName2);
    if (output1 == null || output2 == null) {
      this.input1.despawn();
      this.input2.despawn();
      continue;
    }
    this.finalOutput = createOutput(itemName1, itemName2,
    output1, output2);
    return true;
  }
  return false;
}
```

Figure 3-8: Pseudocode for simplified version of the input generation step

cannot be generated (if it has been spawned to max capacity already or it cannot possess the desired properties), the building block returns a failure. Depending on the building block, it might not always attempt to spawn the output item. For instance, a "Property Change" puzzle will attempt to have its inputs generate the output item while it merely generates a property change relationship.

Step 2: Attempt to generate the input

Summarized in Figure 3-8, after the desired output item has successfully been generated, the building block then constructs a list of items that might function as inputs to the building block. Each building block decides on this list differently. For instance, a combine puzzle will only consider items that can be combined to form the

```
public PuzzleOutput createOutput(String itemName1,
String itemName2,PuzzleOutput output1,
PuzzleOutput output2) {
  PuzzleOutput output = new PuzzleOutput();
  output.Items = concat(output1.Items, output2.Items);
  output.Relationships = concat(output1.Relationships,
  output2.Relationships);
  Relationship rel = buildRelationship(itemName1, itemName2,
  this.outputItem);
  output.Relationships.Add(rel);
  return output;
}
```

Figure 3-9: Pseudocode for simplified version of the relationship generation step

output item. A property change puzzle chooses a specific property it hopes to change and only considers items that have that property or can change that property. Once a list of possible inputs has been produced, the building block shuffles the list and chooses items one by one. For each chosen item, it asks its input building blocks to attempt to generate that item as their output item. It any of its input blocks fail, the building block will move on to the next set of items in its list. If it runs out of items to try, the building block returns a failure. During this step, the building block will also ensure that any input items that need to be transported to the same location are actually "carryable". Items are "carryable" if they possess the "carryable" database property (indicating that the player can carry the item) or one of the inputs to the building block descends from an "Insertion Puzzle" that ensures a container for an non-carryable item will be generated.

Step 3: Build the Relationships

Summarized in Figure 3-9, once all of its inputs manage to successfully generate input items, the building block has succeeded in generating a puzzle and proceeds to generate relevant relationships. Using the input and output items generated in steps 1 and 2, the building block will create relevant relationships determined by the type of the building block. For instance, a combine puzzle block will generate a combine relationship. The building block has now generated a complete puzzle. It returns a list of items and relationships to spawn in the game that includes all items and relationships returned by its input blocks.

3.5 Solvability Guarantees of Puzzle Generator

The puzzle generator algorithm guarantees solvability by induction on the building blocks. For a given block, successfully producing an output item indicates a guarantee that there is a way for the player to acquire that item by making use of relationships and other items in the game. Since each input block also provides this guarantee, then by induction the entire puzzle produced by the generator must be solvable, i.e. the player can acquire the final item representing a successfully solved puzzle. The solvability arguments for each of the building blocks is provided in Appendix B.

Additionally, since each relationship is tied to uniquely spawned items in the game, if the developer of an integrating game only implementes relationships returned by the generator, then the player will be incapable of placing the game in an unsolvable state because only relationships that advance the puzzle can actually be applied. This means that in one game, the generator might spawn a "Hydrogen" item intended to be combined with "Oxygen" to produce "Water". In another game, the generator might generate "Hydrogen" intended to be given to an NPC for a reward, meaning that "Hydrogen" and "Oxygen" cannot be combined to form water in this particular game. Developers who desire consistent rules (i.e. it should always be possible to combine "Hydrogen" and "Oxygen" to form "Water") can spawn additional relationships not specified by the generator. To do so, they would simply construct additional relationships after puzzle generation and add those relationships to their game as appropriate (an example of this process is outlined in 6.1.2). However, these developers must do so with the knowledge that these auxiliary relationships might make it possible for a player to place the game in an unsolvable state.

Despite these solvability guarantees, an important note is that the algorithm described above is an approximation algorithm. The implications of this are that there exist situations where a given database and puzzle map are capable of producing a valid set of puzzles but the *puzzlegen* algorithm might still return a failure. This has an unfortunate effect on usability as a designer might be surprised to discover that a set of inputs that work occasionally will not always successfully generate a puzzle. To cope with this unreliability, designers should expect to invoke the puzzle generator multiple times in the event of failure and also ensure that their databases are flexible enough to produce multiple solutions. Increasing the "capacity" of database items (see Table 4.1) can also improve the chances of a solution being found.

Ultimately, the use of this approximation algorithm is justified by the fact that the problem of determining whether a given set of inputs to the generator can successfully generate a puzzle is actually NP-Complete. This claim is proven in Appendix A and represents one of the major limitations of the system discussed in Chapter 7.

Chapter 4

Database Editor

This chapter describes the item database and the database editor tool used to construct it. The database was designed to have a very simple interface and implementation to increase usability.

4.1 Database Structure for Puzzle Generator

This section describes the structure of the database as seen by the puzzle generator. As the item databases are expected to be fairly small (compared to databases used to store user accounts for web services) and the system isn't concerned about failures or concurrent access, I avoided using a powerful relational database system such as MySQL or Postgres and instead chose to build my own simple database structure. This structure is described below.

4.1.1 Database Items and Properties

The database can be thought of as a single table where the rows of the database are items and the columns are properties. The database provides functions to easily access these items and properties at any time during the puzzle generation process. As described in Chapter 3, properties of items are used by the generator and the integrating game to define how the items are used to generate puzzles and how they

classname	A text property representing the **unique** name of the item. No two items in the database should ever have the same classname.
carryable	A boolean property that determines whether the item can be carried by the player. Necessary for ensuring the solvability of puzzles produced by the generator.
capacity	An integer property that determines how many times an item can be spawned by the generator during puzzle generation. To ensure solvability, each spawn of an item must be uniquely identifiable (see "spawnIndex") with this unique identifier taken into account by the relationships.
abstract	A boolean property indicating whether the item can be spawned within the game. Abstract items will be ignored by the puzzle generator and can be useful for implementing *database extensions* (described in Section 4.1.3).
spawnIndex	An integer property used throughout the puzzle generation process to distinguish two items of the same name. To guarantee solvability, two items of the same name with different purposes should not share relationships, so the "spawnIndex" property is used to distinguish them.

Table 4.1: Common properties expected by the puzzle generator.

behave in the game after they have been spawned. There are a common set of properties that the puzzle generator expects every item in the database to have. These properties are summarized in Table 4.1.

4.1.2 Database Functions

Functions provided by the top-level database object are summarized in Table 4.2. The database maintains two separate structures for items marked "abstract" and normal items. By default, most of the database functions will ignore all items marked "abstract".

Item objects returned by database functions possess their own set of functions, summarized in Table 4.3. These functions expose item properties and whether a database item is free to be spawned.

These database functions are used by database extensions to transform an existing

addExtension	Adds an extension (see 4.1.3) to be run later.
runExtensions	Runs all extensions added to the database in the order they were added.
addItem	Adds a fully formed item to the database.
itemExists	Returns whether an item with a given "classname" exists. Ignores items marked "abstract".
getItem	Retrieves an item by its "classname" from the database. Ignores items marked "abstract". Returns null if it doesn't have a record for the requested name.
itemExistsInMasterLast	Returns whether an item with a given "classname" exists, even if the item is marked "abstract".
getItemFromMasterList	Retrieves an item by its "classname" from the database, even if the item is marked "abstract". Returns null if it doesn't have a record for the requested name.
getItemsWithProperty	Returns a list of item names corresponding to items that possess a certain named property.
filterListByProperty	Given a list of item names, creates a new list that contains only the items that possess a given property. Often used by individual building blocks to find appropriate input items.
copyItem	Given the name of an item to copy and a new name, attempts to copy an existing database item into a new name. Fails if the initial item does not exist or the new name is already present in the database. Primarily used by door unlock puzzle blocks to copy canonical keys (see B.7).

Table 4.2: Functions provided by the whole database.

Spawned	A boolean function indicating whether this item has been spawned to capacity, indicating that it should no longer be spawned in the game.
spawnItem	Increments the spawn reference count for this item and returns a reference to the spawned puzzle item.
despawnItem	Decrements the spawn reference count for this item.
setProperty	Sets a given property to a given value. Primarily called when the database is initially constructed, when an item is being copied, and when a database extension transforms the database.
propertyExists	A boolean function indicating whether this item possesses a value for a given property.
getProperty	Returns the value for a given property, assuming it exists. Returns null if this item does not possess the property.
copyToNewName	Creates and returns a copy of this database item with a new "classname" provided by the caller. Called by the top-level database function "copyItem" and primarily used by door unlock puzzle blocks.

Table 4.3: Functions provided by each object representing a database item.

database and used by the puzzle building blocks to retrieve information about the current state of the database. As database item spawn counts are incremented every time a puzzle is generated, the database must be appropriately reset each time a new puzzle is generated.

4.1.3 Database Extensions

The system allows a developer to extend the database or generate certain portions of the database through the use of "Database Extensions".

A database extension is a simple function run on the in-memory database after it has been constructed from a file exported by the database editor. It can modify the database by adding/deleting items and modifying the properties of items in the database. Database extensions proved incredibly useful while building the sample game described in Chapter 6.

Additionally, features of the database that were previously built-in (such as a flexible inheritance system for database items) were rewritten entirely as database extensions, a move that vastly simplified the code for the database. A few examples of database extensions are included below.

In addition to adding features to the database, Database extensions can be used to partially or completely procedurally generate entire databases via any scheme imagined by the developer, vastly increasing the generality of the database editor system.

Example: Container Size Extension

As described in Appendix B, container puzzle blocks built out of insertion and unboxing blocks expect items to have "filledby" and "fills" properties to indicate which items can be used as containers of other items. The use of these properties was chosen for greater generality (e.g. the designer may wish that the "laptop case" item only be able to hold the "laptop" item), but negatively affects usability of the database editor.

Usability is affected negatively by the use of these properties because filling out all the “filledby” and “fills” properties for every item in the database is a long and repetitive task that can be very prone to error. A much simpler way of thinking about container puzzles would be to have some items specified as containers and to give every item an integer “size” property such that an item can fit inside another item only if it is smaller.

With the introduction of database extensions, it was easy to implement this style of container specification as an extension. Because highly general container relationships were not used in the sample game described in Chapter 6, container relationships were solely expressed using the “size” property and this extension (known as the “FilledbyExtension”). This extension allows a developer to retain both the generality of the original properties and the usability of the new properties.

Example: Changes Extension

As described in Section B.4, a property change block expects database items to have a “changes” property formatted as a dictionary mapping property names to lists of property values. Unfortunately, creating a property with this format is not possible nor necessarily desirable with the current version of the database editor tool. To handle this, a previous version of the database code would automatically convert the format of the “changes” property from a list of string pairs to the expected dictionary.

With the introduction of database extensions, this code was easily ported to an extension (referred to as a “ChangesExtension”), a step that simplified the core database code considerably. When run, the extension simply iterates over all of the database items and modifies their “changes” property to the expected format.

Example: “Parent” Extension

The value of some properties in the database might often be shared across multiple items. For instance, all humanoid items might share the same set of “mutable” properties used by property change blocks. In this case, it would harm usability to require the designer to enter the same set of property values for all items that share the value. This is where an “inheritance” system for the database becomes exceedingly useful. With an inheritance system, items in the database can inherit property values from other items, creating a hierarchical type system that improves the usability of the database editor for the designer.

In an earlier iteration of the tools, this inheritance system was built into the database code, a design that caused a variety of problems. One of the problems with this design was that it forced a single inheritance system on developers who might want a system more appropriate for their needs. For instance, developers might want a child item to inherit certain property values from one item and other property values from another item, something that was not possible with my “one-size-fits-all” inheritance scheme.

Another problem with this design was that it added a lot of messy special case code to the core database. Many properties such as “types” and “parent” were augmented with special meaning and simple functions such as “getProperty” became bloated with handling all possible inheritance cases and breaking out of inheritance loops.

Database extensions presented a solution to both of these issues. By redesigning the inheritance system completely as an extension, I was able to remove all special case inheritance code from the database core, returning it to a simple implementation. Additionally, if the developer wishes to create a new inheritance system or extend the example system I’ve provided (in the “ParentExtension” class), they can simply create a new database extension and replace the existing extension with it.

4.2 Structure of Database Editor

The database editor as it currently exists has been through three iterations. The initial database editor was constructed in the Spring of 2011 by myself as part of a team of undergraduates led by Clara Fernandez-Vara. During the summer of 2011, the team implementing *Stranded in Singapore*[7] with an early version of the tools constructed their own version of the database editor based on the initial version. Finally, in January 2012, I built the current version of the database editor as part of my thesis research using knowledge gained from the previous two iterations.

The editor was designed to be cross-platform in support of generality, so it was implemented as a Java application built with the Java Swing Toolkit. The code is structured using the common “Model-View-Controller” pattern and provides two views, a “List” view where items and properties are displayed as parallel lists and a “Table” view where the entire database is displayed as a table similar to a spreadsheet. Both views were included to support different requirements for entering data into the database.

In both views, a designer can add and delete items and properties to the database via the toolbar at the top of the editor. Items are given a name when added to the database and the editor disallows two items with the same name to exist in the database. This name is automatically added as the “classname” property when the database is exported. Properties are given a name and a type when added to the database. The type of the property determines the GUI editor used to modify it. The property types the editor provides are listed in Table 4.4.

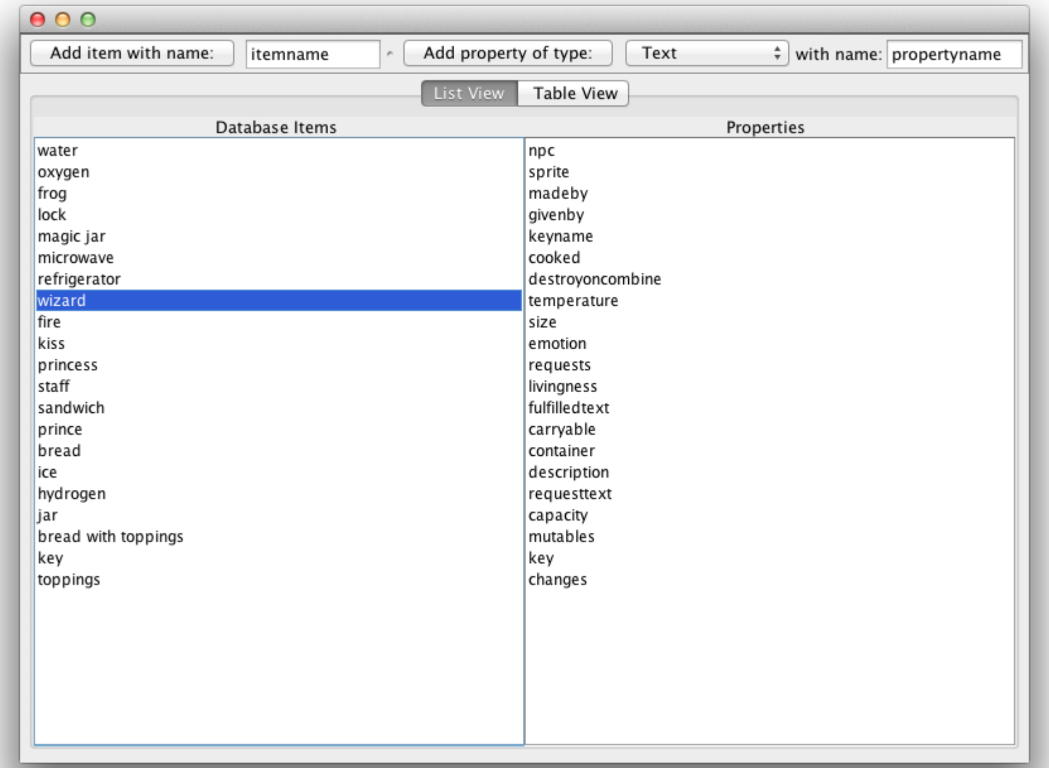
4.2.1 List View

The list view, displayed in Figure 4-1 presents two parallel lists, one for items and one for properties. To modify an item in the list view, the designer simply selects one of the items from the item list on the left. Once selected, that item’s properties can be selected from the list on the right. When a property is selected, it’s value can be modified by a type-specific GUI at the bottom of the editor.

The list view proves useful for designers who want to edit the database on a per-item basis. Whenever a new item is created, the designer can quickly set up all of its properties.

4.2.2 Table View

The table view, displayed in Figure 4-2 presents a spreadsheet-like table view of the entire database. Items are included as rows in this table while properties are

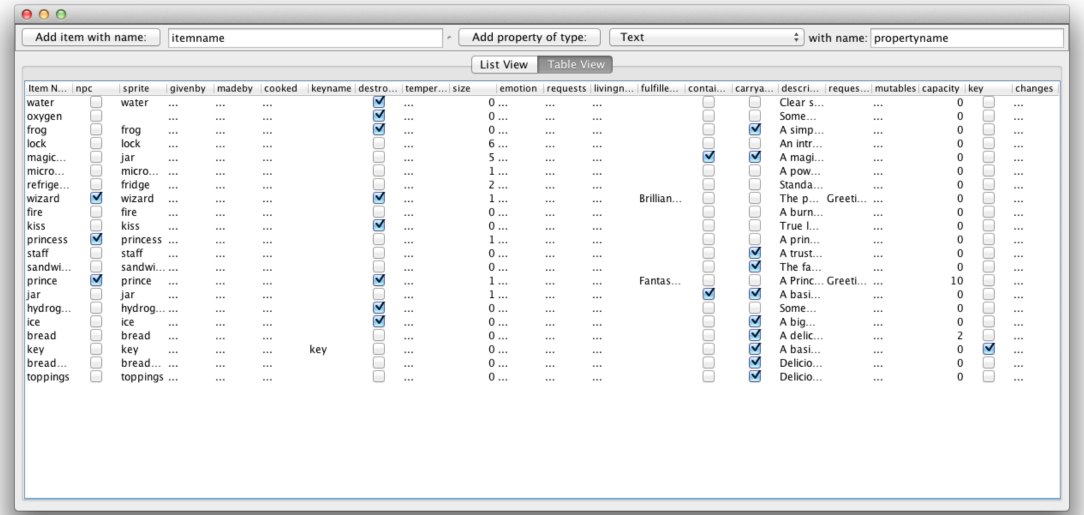


Add item with name: itemname Add property of type: Text with name: propertyname

List View | Table View

Database Items	Properties
water	npc
oxygen	sprite
frog	madeby
lock	givenby
magic jar	keyname
microwave	cooked
refrigerator	destroyoncombine
wizard	temperature
fire	size
kiss	emotion
princess	requests
staff	livingness
sandwich	fulfilledtext
prince	carryable
bread	container
ice	description
hydrogen	requesttext
jar	capacity
bread with toppings	mutables
key	key
toppings	changes

Figure 4-1: A screenshot of the list view of the database editor tool



Add item with name: itemname Add property of type: Text with name: propertyname

List View | Table View

Item N...	npc	sprite	givenby	madeby	cooked	keyname	destro...	temper...	size	emotion	requests	livingn...	fulfille...	contai...	carrya...	descri...	reques...	mutables	capacity	key	changes
water	☐	water	...	...	...		✓	...	0	...	...	...		☐	☐	Clear s...		...	0	☐	...
oxygen	☐		...	...	...		✓	...	0	...	...	...		☐	☐	Some...		...	0	☐	...
frog	☐	frog	...	...	...		✓	...	0	...	...	...		☐	✓	A simp...		...	0	☐	...
lock	☐	lock	...	...	...		☐	...	6	...	...	...		☐	☐	An intr...		...	0	☐	...
magic...	☐	jar	...	...	...		☐	...	5	...	...	...		✓	✓	A magi...		...	0	☐	...
micro...	☐	micro...	...	...	...		☐	...	1	...	...	...		☐	☐	A pow...		...	0	☐	...
refrige...	☐	fridge	...	...	...		☐	...	2	...	...	...		☐	☐	Standa...		...	0	☐	...
wizard	✓	wizard	...	...	...		✓	...	1	...	...	...	Brillian...	☐	☐	The p...	Greeti...	...	0	☐	...
fire	☐	fire	...	...	...		☐	...	0	...	...	...		☐	☐	A burn...		...	0	☐	...
kiss	☐	kiss	...	...	...		✓	...	0	...	...	...		☐	☐	True l...		...	0	☐	...
princess	✓	princess	...	...	...		☐	...	1	...	...	...		☐	☐	A prin...		...	0	☐	...
staff	☐	staff	...	...	...		☐	...	0	...	...	...		☐	✓	A trust...		...	0	☐	...
sandwi...	☐	sandwi...	...	...	...		☐	...	0	...	...	...		☐	✓	The fa...		...	0	☐	...
prince	✓	prince	...	...	...		✓	...	1	...	...	...	Fantas...	☐	☐	A Princ...	Greeti...	...	10	☐	...
jar	☐	jar	...	...	...		☐	...	1	...	...	...		✓	✓	A basi...		...	0	☐	...
hydrog...	☐	hydrog...	...	...	...		✓	...	0	...	...	...		☐	☐	Some...		...	0	☐	...
ice	☐	ice	...	...	...		✓	...	0	...	...	...		☐	✓	A big...		...	0	☐	...
bread	☐	bread	...	...	...		☐	...	0	...	...	...		☐	✓	A delic...		...	2	☐	...
key	☐	key	...	...	...	key	☐	...	0	...	...	...		☐	✓	A basi...		...	0	✓	...
bread...	☐	bread...	...	...	...		☐	...	0	...	...	...		☐	✓	Delicio...		...	0	☐	...
toppings	☐	toppings	...	...	...		☐	...	0	...	...	...		☐	✓	Delicio...		...	0	☐	...

Figure 4-2: A screenshot of the table view of the database editor tool

Text	A simple string property type. Used for properties such as "description".
Integer	An integer property type. Used for properties such as "capacity" and "size".
Boolean	A boolean property type. Used for properties such as "carryable" and "NPC".
Item List	A property type representing a list of other items in the database. Used for properties such as "givenby", "gives", and "parents" (for the inheritance extension).
String List	A property type representing a list of strings. Usually used to create a list of property names for properties such as "mutables" and various mutable properties.
String Pair List	A property type representing a list of string pairs. Effectively used whenever a property requires pairs, such as with the "madeby" property. Also used for the "changes" property before it is converted by an extension.
Custom	A user defined property type that can include any number of sub-properties which can have any property type except "Custom". When this type is selected, an editor window pops open allowing the designer to create a custom property. The puzzle generator treats a custom property as a dictionary mapping strings (sub-property names) to objects (sub-property values).

Table 4.4: The property types provided by the database editor.

columns. Basic property types such as "Text", "Integer", and "Boolean" types can be modified directly in the table cells. More complex property types such as "Item List" or "Custom" types display a "..." in their cells that, when selected, pops open an appropriate editing GUI at the bottom of the window.

The table view proves useful for designers who want to edit the database on a per-property basis. For example, when a new boolean property is added to the database, the designer can simply switch to the table view and quickly set all the check boxes as appropriate for each item in the database.

4.2.3 Output Format of Database Editor

The database editor exports to a simple XML format that specifies every item and every property. Properties are described by their name, type, and default values. Each item is listed with a list of the properties it possesses a value for along with the corresponding values. This output format was chosen to be human readable and cross-platform. The puzzle generation library has a simple parser that produces an in-memory database object when given an XML file containing the database editor output.

4.3 Future Development of Database Editor

While the database editor has proven very powerful and usable for building item databases, there still remains some possible improvements for future iterations.

For instance, the current database editor provides types such as “String Pair List” and “String List” that are commonly used to create lists of properties, property pairs, and item-property pairs. Future development might take these uses into account and construct specific property types tailored to these uses.

Additionally, the database editor currently lacks common functionality expected of a graphical editor, most notably an “Undo” function for correcting mistakes. While this is a desirable feature, I chose to keep it low priority because all the actions are already reversible within the editor. Future development of the editor could implement this and other desirable features that improve usability of the editor.

Chapter 5

Puzzle Map Editor

This chapter describes the graphical tool used to construct the puzzle map inputs to the puzzle generator. This tool has been through only two iterations, fewer than the puzzle generator and the database editor, because it was a later addition to the Puzzledice system. In the original version of the system, puzzle maps were created in code by manually instantiating objects.

During the summer of 2011, the team that developed *Stranded in Singapore*[7] decided that a level editor was necessary and constructed an editor that was capable of building levels for their game as well as puzzle maps for the Puzzledice system. As part of my continuing research, I iterated on their design and constructed a new puzzle map tool designed to meet the goals of generality and usability.

5.1 Structure of Puzzle Map Editor

Similar to the database editor, the puzzle map editor was designed to be cross-platform in support of greater generality, so it was also implemented as a Java application built with the Java Swing Toolkit. The editor makes use of the open source *JGraph* library to display both an undirected area graph in an "Area View" window and a directed puzzle graph in a "Puzzle View" window. Both areas and building blocks can be named to improve readability of the graphs with the restriction that no two area blocks or puzzle blocks can share the same name.

Along with the visual representations of the puzzle maps provided by the two views, the puzzle editor presents a "puzzle description" as a short textual description representing the kind of puzzle that would be produced by this map (with placeholder names for items).

Regardless of the view, a designer can add and delete areas, connections between areas, puzzle blocks, and connections between puzzle blocks via the toolbar at the top the window. This toolbar also allows the designer to edit certain aspects of specific building blocks (e.g. choosing a specific property and value for a property change block) and mark which of the area blocks is intended to be the start area of the map (a start area is always necessary for a valid puzzle map).

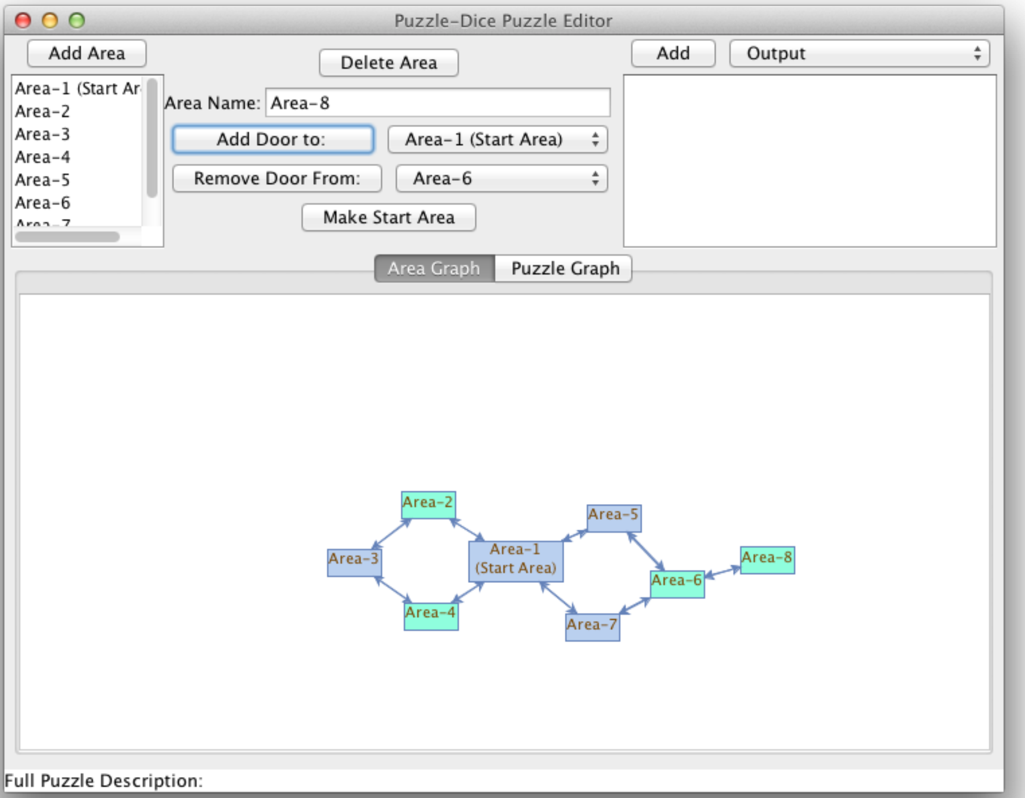
5.1.1 Area View

The "Area View" window presents a current view of all the areas blocks in the map and direct connections between them. The area view presents an undirected graph of area blocks. Locked doors are not included in the area view as they rely heavily on the directionality of the puzzle map and are therefore not correctly represented by an undirected graph.

While it might seem unrelated to the actual puzzle map, the area view and graph proved useful from a usability standpoint as designers naturally like to think about areas and connections between them in this way (as opposed to generating all area blocks in the puzzle view and manually choosing their input areas). The area view provides generality by treating an area map as a general undirected graph, leaving the actual structure of the in-game area map up to the developers.

5.1.2 Puzzle View

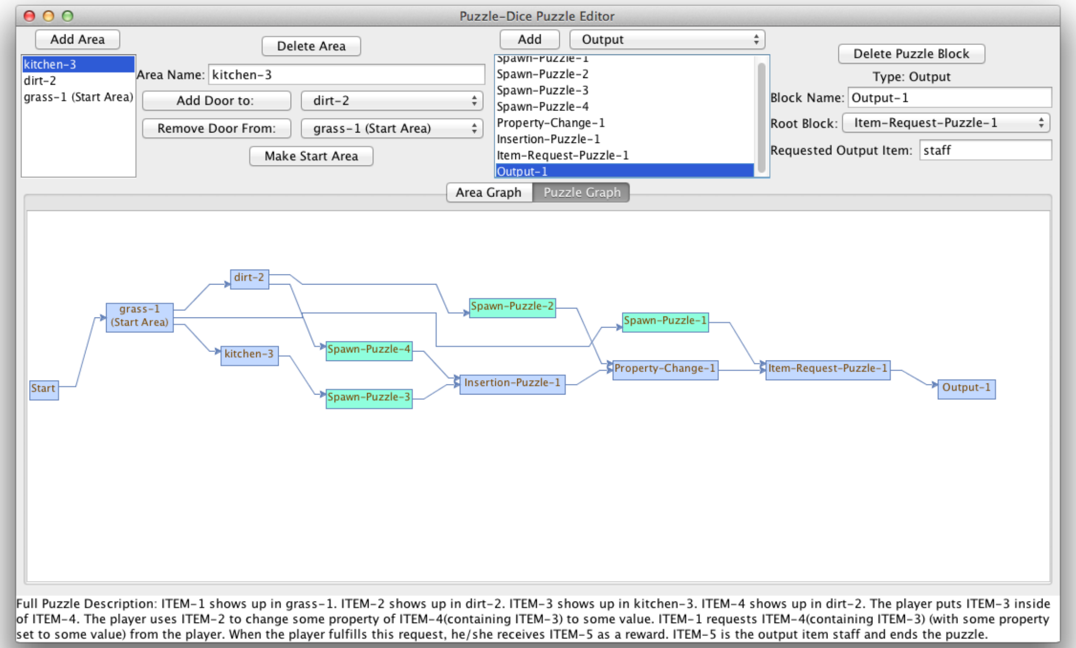
The "Puzzle View" window presents a current view of the puzzle map (including both area blocks and puzzle blocks) as a directed acyclic graph. The graph shown by this view is very similar to the figures used throughout this document to represent puzzle maps.



Area Name: Area-8

Full Puzzle Description:

Figure 5-1: Screenshot of the area view of the puzzle map editor



Area Name: kitchen-3

Add Door to: dirt-2

Remove Door From: grass-1 (Start Area)

Type: Output

Block Name: Output-1

Root Block: Item-Request-Puzzle-1

Requested Output Item: staff

Full Puzzle Description: ITEM-1 shows up in grass-1. ITEM-2 shows up in dirt-2. ITEM-3 shows up in kitchen-3. ITEM-4 shows up in dirt-2. The player puts ITEM-3 inside of ITEM-4. The player uses ITEM-2 to change some property of ITEM-4(containing ITEM-3) to some value. ITEM-1 requests ITEM-4(containing ITEM-3) (with some property set to some value) from the player. When the player fulfills this request, he/she receives ITEM-5 as a reward. ITEM-5 is the output item staff and ends the puzzle.

Figure 5-2: Screenshot of the puzzle view of the puzzle map editor

Because the puzzle map takes the area graph into account (by including some areas as inputs to other areas), the puzzle graph will often rearrange itself in response to changes of the area topology. These changes can include adding or deleting area blocks and connections or changing which area is marked as the "Start Area".

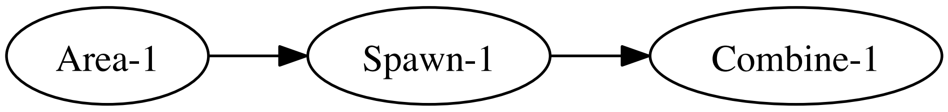


Figure 5-3: An incomplete puzzle map that can still produce a text description

5.1.3 Puzzle Description

The puzzle map editor provides a textual description of the general flow of a puzzle map via the "puzzle description" included at the bottom of the window. To view a puzzle description, the designer selects one of the puzzle blocks from the list at the top and a description appears at the bottom.

The descriptions are general and specific to the building blocks used in the map. When a block does not yet have an input selected, the input is listed as "SOMETHING" or "SOMEWHERE" in the description. For example, the puzzle map shown in Figure 5-3 would have the following description: "ITEM-1 shows up in Area-1. The Player combines ITEM-1 with SOMETHING to create ITEM-2."

5.1.4 Output Block

One building block available in the puzzle map generator that is not available as an in-memory building block is the "Output" block. This block is used to indicate the starting point for generating the puzzle map when the runtime library is parsing the output of the editor. Without any of the blocks being marked as the "Output", the generator has no way of knowing which block marks the end of the puzzle and will be unable to produce a valid in-memory puzzle map.

In the editor, the input to the "Output" block is simply the puzzle block that the designer intends to represent the output of the map. The designer also has the option to specify a specific item name to generate as output but this feature is currently ignored by the simple parser. When parsing the editor output, the parser will choose the first "Output" block it finds as the output to the puzzle, so in most cases there will only be one such block.

5.2 Valid Puzzle Map Guarantees

For usability purposes, it is important that the puzzle editor only allow a designer to construct puzzle maps that the system considers "valid". For a puzzle map to be valid, it must be acyclic (so the generator does not enter an infinite loop) and building blocks that require certain types of inputs (such as spawn puzzles that require an area block as input) must only have those types of block as their inputs.

The latter requirement is satisfied by the editor separating area and puzzle blocks into two separate lists (and UI sections). Puzzle blocks that require an area block as input will only present the list of area blocks as valid choices while puzzle blocks that require other puzzle blocks as input do likewise.

The former requirement is a little trickier. Normally, if the designer directly attempts to introduce a cycle into the graph (e.g. an attempt to attach the output of a combine block as its input), the editor will detect this, inform the designer of the mistake, and disallow the operation. Unfortunately, since the area blocks and puzzle blocks are separated and the the editor attempts to build an acyclic puzzle map based on the area topology, cycles can be subtly introduced by seemingly innocuous user actions. If the topology of the areas is changed by the user, it is entirely possible the change will introduce a cycle into the puzzle map. The editor handles this by first informing the designer that the topology change introduced a cycle and then carefully breaking connections as necessary to remove the cycle from the graph. While this solution is not very elegant, it is very explicit to the designer about when cycles occur.

5.3 Output Format of Puzzle Map Editor

Like the database editor, the puzzle map editor exports to a simple XML format. This format specifies each of the areas (along with their connections) and each of the building blocks with appropriate inputs. This output format was chosen to be human readable and cross-platform. The puzzle generation library has a simple parser that constructs an in-memory puzzle map object (represented by a single recursive building block) when given an XML file containing the puzzle map editor output.

An important note is that while the simple parser will take *Area* names into account when parsing the XML (i.e. it will only generate one block corresponding to "area-1"), it will construct a new block for every puzzle block it encounters regardless

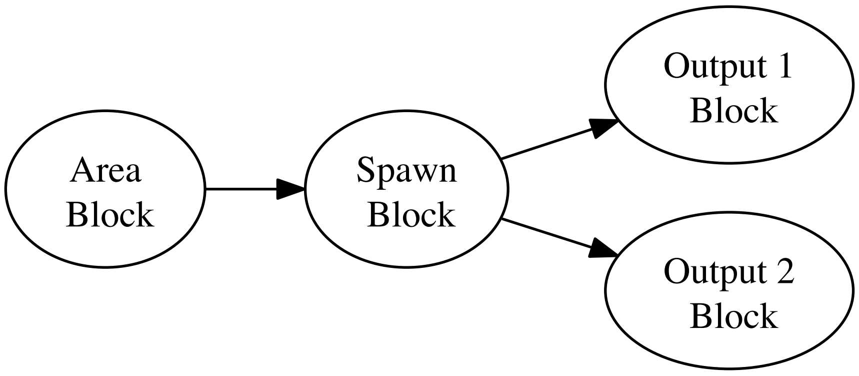


Figure 5-4: A normally illegal puzzle map (because the spawn block has two outputs)

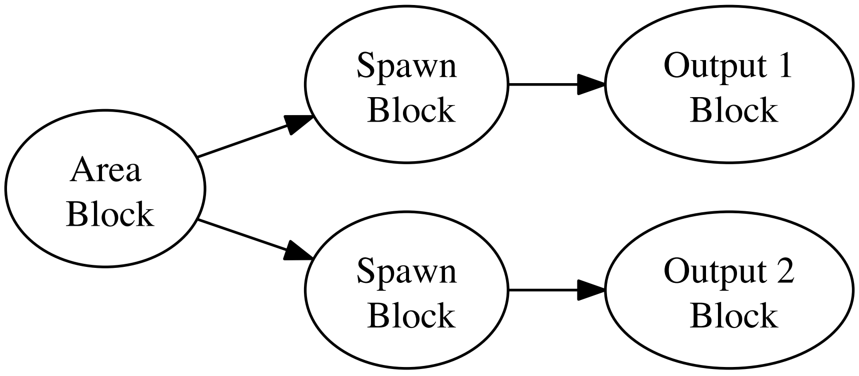


Figure 5-5: The puzzle map parser will split the map in Figure 5-4 into this legal version

of whether it has seen the name already. It does this because in a valid puzzle map every puzzle block must only have *one* output block, but a puzzle map that ignores this requirement can easily be converted into a valid puzzle map by splitting repeated puzzle blocks into multiple blocks.

Figure 5-4 and Figure 5-5 represent how the parser of the puzzle map's output converts seemingly illegal puzzle maps (as in Figure 5-4) to a corresponding legal map (as in Figure 5-5). This effect creates greater usability for the designer as fewer puzzle blocks need to be shown on screen for the same puzzle maps. For instance, I found that a common work flow for creating puzzle maps for the sample game (described in Chapter 6) was I would create a single spawn block for each area and then attach multiple blocks as output as needed. I found that this technique massively reduced clutter and the speed with which I could construct the puzzle maps.

5.4 Future Development of Puzzle Map Editor

While the puzzle map editor has proven very powerful and usable for building puzzle maps, there still remains some possible improvements for future iterations.

Most notably, there is currently no facility to include custom designed building blocks in the puzzle map editor. A future iteration might include a special type of building block (called "Custom") that pops open a window where the designer can name the various inputs to the block. This would allow developers to include their own custom-made building blocks in the puzzle map editor and would vastly improve the generality of the system.

Another possible improvement is the graph representation of the area view and puzzle view could certainly stand some improvement. Due to the rather finicky nature of the *JGraph* library, the animating graphs occasionally succumb to visual glitches or artifacts. Future development might wish to fix these to improve the aesthetics of the editor.

Additionally, the puzzle map editor currently lacks common functionality expected of a graphical editor, most notably an "Undo" function for correcting mistakes. While this is a desirable feature, I chose to keep it low priority because all user actions are already easily reversible within the editor and a usable implementation of “Undo” can be difficult to design well. Future development of the editor could implement this and other desirable features that improve the usability of the editor.

Finally, the current implementation of the puzzle map editor relies entirely on the toolbars at the top of the window for editing while the graph views are merely a visual representation of the puzzle. A more ambitious UI might allow a designer to construct puzzle maps completely by interacting with the graph view and adding input and output connections as needed. While designing such a UI was beyond the scope of my thesis, it remains as an interesting challenge for future work.

Chapter 6

Evaluation

To evaluate the usability and generality (and solvability to a lesser extent) of the system, I constructed a sample game to demonstrate how the system could be easily integrated with existing game development tools. This section describes the steps necessary to integrate the sample game with the puzzle generator and includes my thoughts on how well the system achieves its goals.

The complete source code for this sample game “Sandwitch” (with proprietary rendering components removed) is included with the rest of the system and can serve as a starting point for any developers interested in integrating the system with their own games.

6.1 Description of Game

The sample game, called “Sandwitch”, is a simple 2D tile-based “Adventure” game featuring a fantasy setting and basic graphics and sound. The player controls a witch character with the keyboard and is able to move around the game world and interact with various items and creatures. The goal of each play-through is to find a “sandwich” item and place it in the player’s inventory. Doing so involves exploration, performing tasks for NPCs, using items with each other to alter properties or create new items, and unlocking doors.

The game was built with the Unity game engine and the proprietary *2DToolkit*

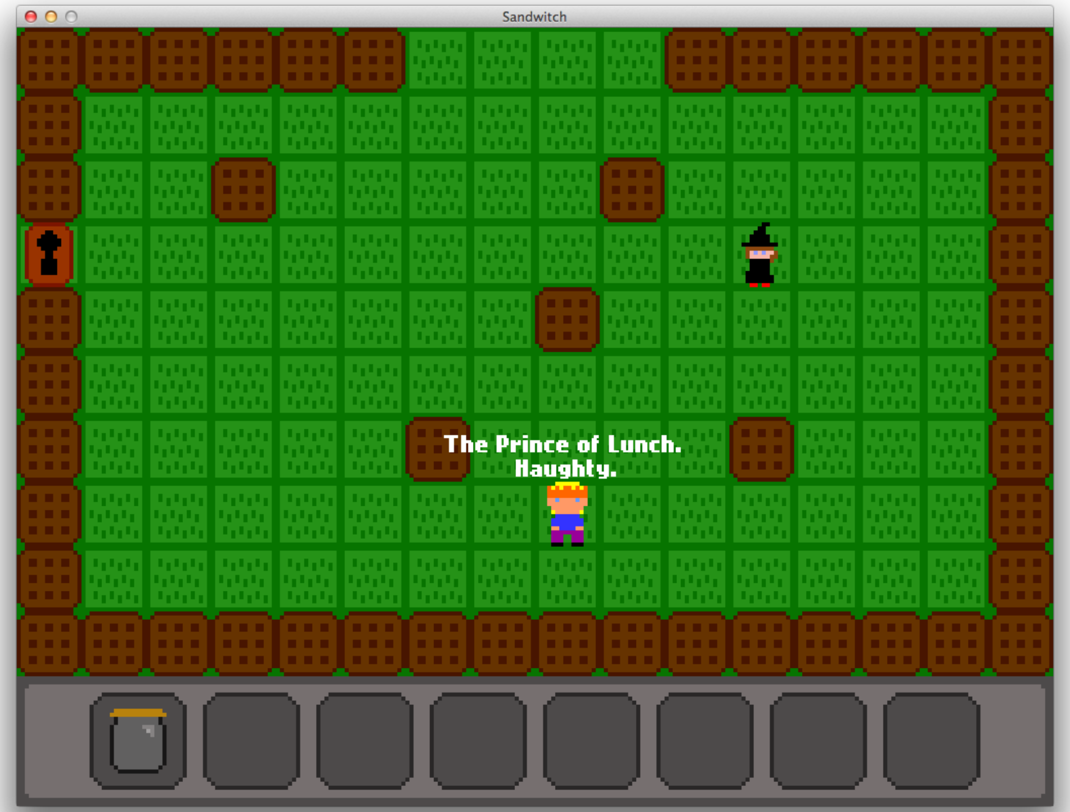


Figure 6-1: Screenshot from "Sandwitch", a sample game made to present the capabilities of the system

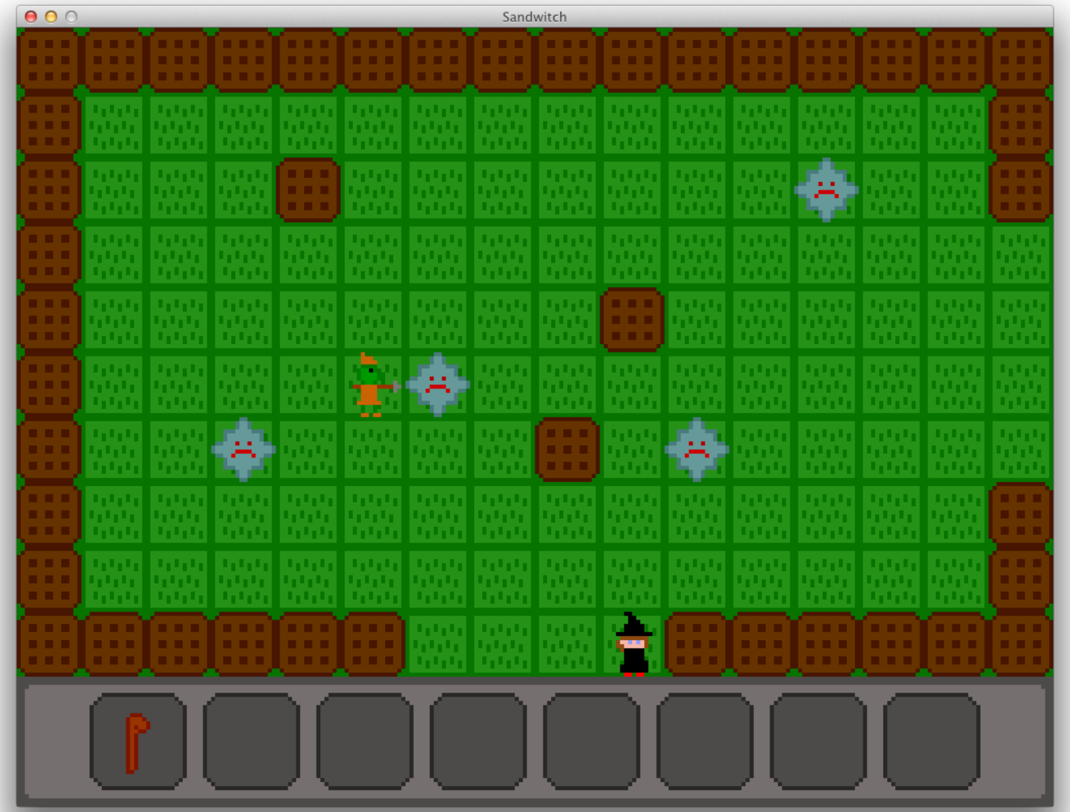


Figure 6-2: Screenshot of a filler room filled with enemies

library for easily managing 2D objects. As I cannot include the source code for *2DToolkit* in my public release, all *2DToolkit* code has been removed from the project. References to the affected components were left in the code so that interested developers could easily compile and run the game after purchasing a license for the *2DToolkit* library and including the library in the project.

The game features two components that were used to experiment with the solvability and generality of the Puzzledice system: filler rooms and auxiliary relationships. These components are described in more detail below.

6.1.1 Filler Rooms

To show how the Puzzledice system can be integrated with games that feature other kinds of mechanics and puzzles than the ones generated by Puzzledice, I included a set of "filler rooms" between the rooms that corresponded to area blocks in the puzzle map.

These rooms contain simple enemies that impede the player's progress. Ultimately, these enemies pose no threat to the player; they are merely there to expose how traditional game elements can be added to a game integrated with Puzzledice. A screenshot of one of these filler rooms is shown in Figure 6-2.

6.1.2 Auxiliary Relationships

As mentioned in Chapter 3, the puzzle generator maintains perfect solvability by generating *only* the items and relationships necessary for solving the puzzle. Unfortunately, this can make games inconsistent. For instance, in one game, the "Hydrogen" and "Oxygen" items are combined to form "Water". In another game, "Hydrogen" and "Oxygen" are produced independently as the request items of certain NPCs. In the second game, if the two items could be combined to form "Water", the player might be able to put the game in an unsolvable state even though this combination is consistent with the rules of the first game.

To experiment with relaxing solvability requirements, I added "Auxiliary Relationships" to Sandwitch. These are relationships that exist in the database but aren't necessarily produced by the generator as part of the solution. They allow for consistent behavior of items across games at the expense of perfect solvability. A warning was added to the instructions screen of Sandwitch to acknowledge that players *can* place the game in an unsolvable state if they are not careful and that they should reset the game if they feel they have done so.

A more sophisticated use of auxiliary relationships would detect what relationships can be generated without violating perfect solvability but these go beyond the scope of my research. The auxiliary relationships in Sandwitch are merely there to present how solvability can be relaxed to create consistency.

6.2 Integration Process

This section outlines the steps that were necessary for integrating "Sandwitch" with the puzzle generation library. The major task for this integration was converting the abstract outputs of the puzzle generator into concrete objects and rules in the world of the game. For Sandwitch, this task consisted of three parts, each outlined below.

6.2.1 Spawn Puzzle Items

The most obvious task for integration was taking the abstract item descriptions from the puzzle generator output and producing their in-game representations. This task was relatively simple. Puzzle objects were each represented by an instantiated Unity prefab called "puzzleitem". This prefab has a *2DToolkit* sprite component and a script component called "SpawnedPuzzleItem" which maintains properties for the item.

When an item generated by the puzzle generator is added to the game, a new "puzzleitem" is instantiated and all of the properties of the item are copied to the "SpawnedPuzzleItem" component. One of these properties is called "sprite" and its value corresponds to an ID of one of the sprites in the *2DToolkit* sprite atlas. The "SpawnedPuzzleItem" component tells the *2DToolkit* sprite component which sprite to use based on this property.

Other properties of the puzzle item, such as "carryable" or "NPC" define the behavior of the "puzzleitem" object. For instance, if the item is not marked as "carryable", the player treats it like a wall and cannot pick it up. If the item is marked "NPC", then text provided by other properties is displayed when the player bumps into the object.

6.2.2 Integrate Relationships

The next major task for integration is to include the abstract relationships produced as output of the puzzle generator and integrate them into the logic of the game. For

Sandwitch, the “AreaConnection” and “StartArea” relationships were kept separate from the rest of the relationships as they mostly pertained to area generation.

To add custom behavior to the game for each type of relationship, a “Relationship Visitor” was constructed that defined a special function for each of the relationships. Using the “Visitor” pattern, each relationship produced by the generator is sent to this visitor object and added to the appropriate data structures.

Most relationships in the game are added to a “Relationship Map”. This structure maps the ids of pairs of items to a possible relationship between them. When the player attempts to use two items together, their ids are sent to the relationship map. If a relationship was found to exist between the two items, this relationship is passed along with the items to a “RelationshipExecutor” (itself another relationship visitor) that actually performs the effect of the relationship while taking item properties into account.

As an example, when two items that can be combined are used together, the relationship map returns a “Combine” relationship which the executor than applies to the game by spawning the result item and (possibly) destroying the two ingredient items. In Sandwitch, the executor only destroys ingredients that have the “destroyOnCombine” property set to true. This is one example of using database properties for purposes beyond puzzle generation.

Auxiliary relationships are added to the relationship map after the relevant puzzle relationships are added. The game looks through the relationship map to find pairs of items that do not share a relationship. It then attempts to construct a relationship based on database properties. For instance, it will always construct insertion relationships for containers and items that can fit inside and it will occasionally construct auxiliary combine and property change relationships.

6.2.3 Connect Areas

The last task for the integrating game is to take abstract information about areas and their connections and somehow construct a physical space representing the areas.

This process can be handled a variety of ways depending on the game. In an early iteration of the tools, we built a text adventure integrated with the generator that could keep the area graph in almost exactly the same form with little difficulty. Another possible implementation might make a single "Floor" per area and connect the floors via stairs and ladders.

For Sandwitch, each area is represented by a single screen-sized room that can have four possible exits. The areas are connected by a small set of "filler" rooms (see 6.1.1) that branch randomly from their source rooms. I made sure that none of the areas were given more than four connections in my puzzle maps and that the number of filler rooms was small enough that intersections between area paths was not a problem.

Items are then spawned in the appropriate room as indicated by their "spawnArea" property. Locked doors are placed at the exit of the "source" area in a door unlock puzzle. A relationship is added to the relationship map between the lock object and the spawned key indicating that the key can remove the lock. For convenience, the lock object is included in the database as a database item, but it is marked "abstract" so that no building block attempts to generate it. This case presents another use of the "abstract" tag for database items.

6.3 Solvability of Puzzles

Without auxiliary relationships, I never encountered a situation in which the game produced an unsolvable puzzle while playing. I also never encountered a case where I placed the game into an unsolvable state, indicating that the solvability guarantees of the puzzle generator held true.

Of course, the introduction of auxiliary relationships changed the solvability of the game. While it was still true that the game would never produce an unsolvable puzzle, it definitely became possible to place the game in an unsolvable state if the wrong auxiliary relationship was activated. In my experience, this was relatively rare and it became apparent that the game was unsolvable fairly quickly due to its small size, but the fact remains that auxiliary relationships should be used with caution.

6.4 Generality of System

The entire development of Sandwitch was mostly a test of generality. While previous iterations of the tools had been used to construct text and graphical adventure games, I wanted to show that the system could be integrated with a game that might have other mechanics such as action sequences. The filler rooms outlined in Section 6.1.1 were designed to display this kind of generality.

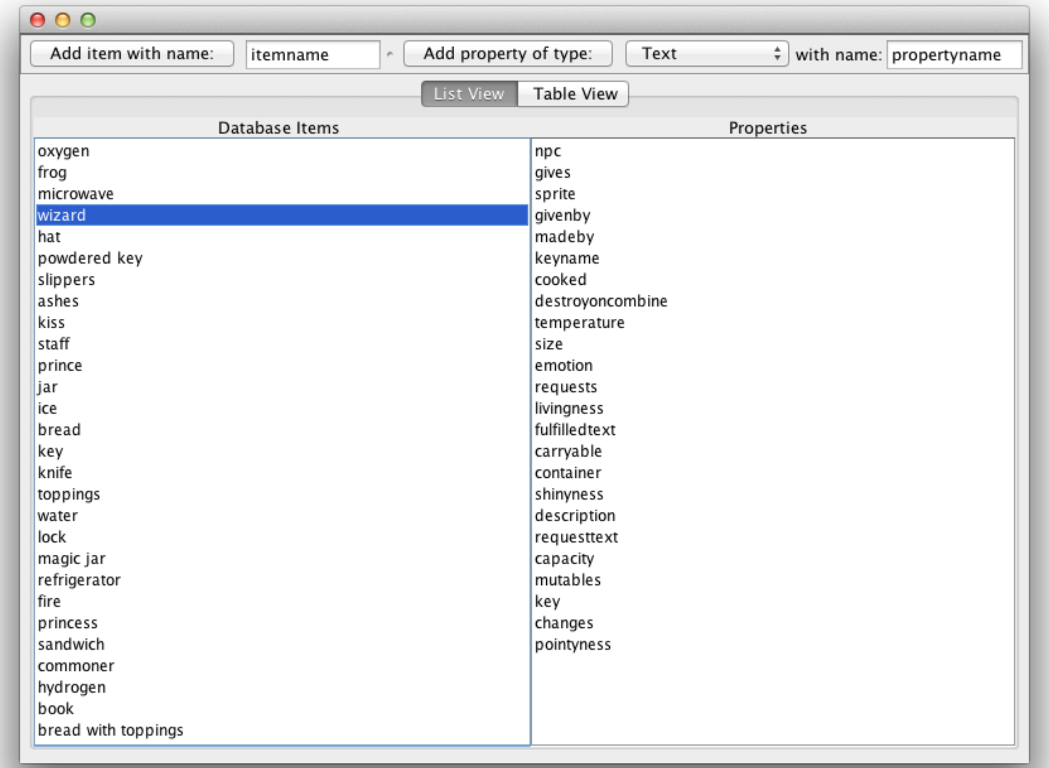
The database editor also proved to be quite useful as a general purpose tool. As mentioned above, game specific logic was stored in database properties and items for convenience. If I had wanted, I could have stored the entire game configuration in the database which is a strong argument for its generality.

6.5 Usability of System

The graphical tools proved very usable for developing the sample game. Figure 6-3 shows a screenshot of the database used by the final version of the game as viewed from the database editor. There were no issues constructing this database with the database editor tool and I found myself using both views depending on the circumstances.

Figure 6-4 shows a screenshot of one of the puzzle maps used by the game as viewed from the puzzle map editor. While the UI of the puzzle map editor was slightly cumbersome due to the use of toolbars and menus, I also experienced no major problems with this tool and was able to construct a wide variety of puzzle maps with little difficulty.

Overall, it was pretty straightforward to integrate the game with the system after the initial game logic and rendering system was set up. The only major speedbumps that occurred when working with the puzzle generator were due to bugs in the implementation which have since been fixed. Overall, I consider the system usable enough that I plan to use it extensively in my future games.



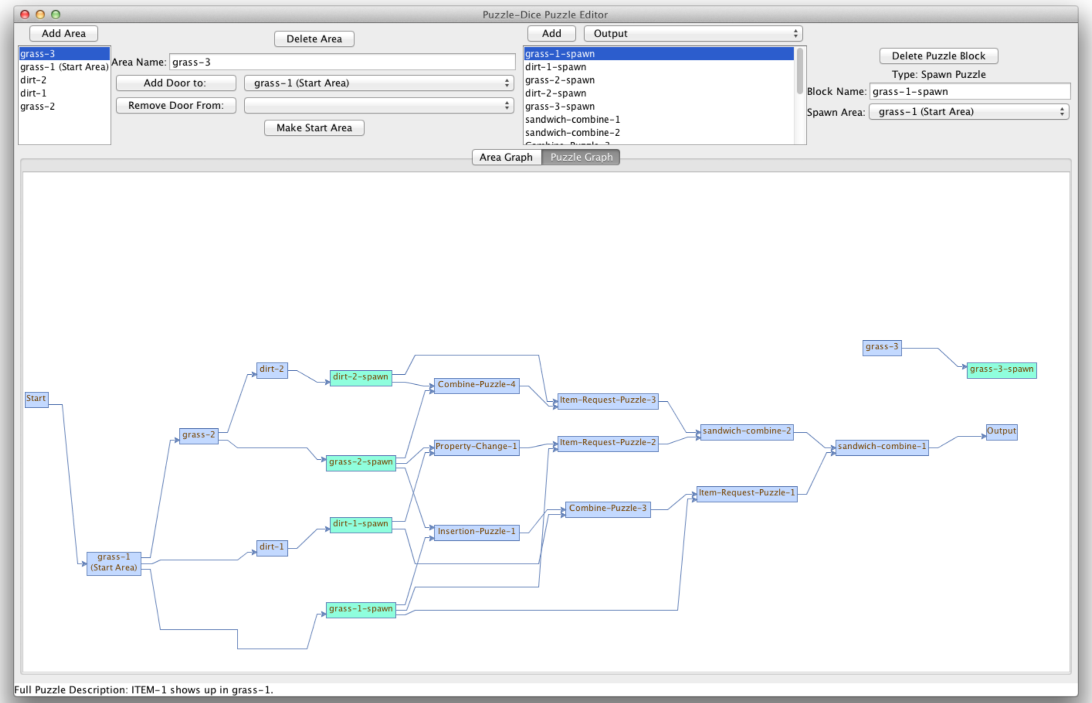
Database Items

oxygen
frog
microwave
wizard
hat
powdered key
slippers
ashes
kiss
staff
prince
jar
ice
bread
key
knife
toppings
water
lock
magic jar
refrigerator
fire
princess
sandwich
commoner
hydrogen
book
bread with toppings

Properties

npc
gives
sprite
givenby
madeby
keyname
cooked
destroyoncombine
temperature
size
emotion
requests
livingness
fulfilledtext
carryable
container
shinyness
description
requesttext
capacity
mutables
key
changes
pointyness

Figure 6-3: The database for “Sandwitch”, as shown in the database editor



grass-3
grass-1 (Start Area)
dirt-2
dirt-1
grass-2

Area Name: grass-3

Add Door to: grass-1 (Start Area)

grass-1-spawn
dirt-1-spawn
grass-2-spawn
dirt-2-spawn
grass-3-spawn
sandwich-combine-1
sandwich-combine-2

Type: Spawn Puzzle

Block Name: grass-1-spawn

Spawn Area: grass-1 (Start Area)

Full Puzzle Description: ITEM-1 shows up in grass-1.

Figure 6-4: One of the puzzle maps for “Sandwitch”, as shown in the puzzle map editor

Chapter 7

Conclusion

7.1 Limitations

The system outlined in this thesis has a number of limitations, each outlined below. These limitations are related to the object oriented approach of the system, the generality of the system, and the inherent algorithmic difficulty of the system.

7.1.1 Narrowness of object oriented approach

The focus of the Puzzledice system is on puzzles involving the manipulation of physical objects represented in game. While this approach is well suited for a variety of puzzles common to games, there are many types of puzzles for which this approach fails.

For instance, it would be difficult for the system as-is to generate puzzles that rely on spatial reasoning (such as block pushing puzzles, sliding puzzles, traditional jigsaw puzzles, etc.) or puzzles that rely on word-play. Puzzles that rely on there being more than one relationship between two objects can also be difficult to implement with this system.

Despite the narrowness of this approach, it can be adapted to puzzles that are seemingly unrelated to objects. For instance, "Hidden Information" puzzles where the player must discover some piece of information (such as a safe combination) before being able to continue could be generated by the system by treating the hidden information as an object in the database and implementing the relationships as appropriate in-game.

Ultimately, the object oriented approach of this system is a limitation but can be worked around in certain situations.

7.1.2 Tradeoff between generality and usability

Because one of the goals of this thesis was to produce a system that could be integrated with a wide variety of games, the system strives to provide as much generality as possible. Unfortunately, this generality makes the task of integrating the system with a game considerably more challenging, ultimately affecting the usability of the system.

Unfortunately, the problem of procedural puzzle generation presents a direct tradeoff between generality and usability. Simply put, the more general the system is, the more details exist that will need to be filled in by a developer integrating the system with a game. To alleviate this problem, the idea of Puzzledice "Front-Ends" is presented below in section 7.2.3.

7.1.3 Algorithmic Difficulty

As proven in Appendix A, the puzzle generation problem being solved by the puzzle generator is NP-Complete, which justifies my use of an approximation algorithm for the puzzle generator. As explained in Chapter 3, the approximation algorithm can occasionally have a harmful effect on the usability of the system.

Ultimately, the more general problem of determining whether a given game is solvable is often NP-Hard, as shown in [16]. Since a common approach for procedurally generating puzzles is to first randomly generate a puzzle and then verify its solvability, these results show that producing interesting and perfectly solvable puzzles and games can often be algorithmically intractable.

Despite this limitation, procedural generation remains a rich field of study. Many games, including [17] choose to relax the solvability requirement in favor of more interesting puzzles. They do so by providing the player with enough “life-lines” that most unsolvable situations can be escaped at the cost of some player resources.

The Puzzledice system allows solvability to be relaxed on an implementation-by-implementation basis, providing a good amount of flexibility for developers. As outlined in Chapter 6 the system will produce perfectly solvable puzzles if only the items and relationships produced by the generator are added to the game. More interesting puzzles and multi-puzzle solutions can be added to a game by including additional relationships as desired.

7.2 Future Work

While the system as presented represents a complete body of work, there are still potential improvements and refinements to the system.

7.2.1 More General Building Blocks

The existing building blocks generate puzzles by relying on the simple names and properties of items described in the database. While this is sufficient to produce basic puzzles, more general building blocks that rely more heavily on database properties could produce interesting results.

For example, the existing “Combine” block will create a Combine relationship that combines two items to produce a third item. This relationship does not impose any restrictions on the properties of the ingredient items, yet this kind of restriction might make sense in the context of a game. For instance, iron and hammer are combined to produce a blade, yet a developer might want the player to first heat up the iron at forge (changing the temperature property) before it can be combined with the hammer. A more general Combine block that imposes these sorts of restrictions might be desirable and is worth looking into.

7.2.2 Puzzle Map Generation

While the puzzle map editor allows a designer to easily specify the shape of puzzles to be produced by the puzzle generator, a system that procedurally generates the puzzle maps could potentially produce more interesting and unexpected puzzles with less effort on the part of the designer.

Additionally, since the database extensions outlined in Chapter 4 can be used to procedurally generate a database, a logical next-step would be to allow similar extensions to the puzzle maps produced by the puzzle map editor. While the OR-Blocks allow for a small degree of dynamic puzzle maps, it would be worthwhile exploring more complex schemes that take database relationships into account when generating puzzle maps.

7.2.3 Front-Ends

One of the original goals of this project was for the tools to be usable by non-programmers. While the graphical tools meet this goal, there still exists a requirement for the runtime library to be integrated with an existing game, a task requiring non-trivial amounts of programming.

To further improve the usability of the system for non-programmers, it is possible and might be worthwhile to create a series of "front-ends", pre-integrated template games that allow designers use only the graphical tools to take advantage of the system. For example, the sample game described in Chapter 6 could be converted into a front-end by building a tool that accepts designer inputs (such as art and puzzle generation parameters) and builds a version of the game using those inputs. Other possible front-ends include an "Interactive Fiction" front-end that produces a text game controlled by a parser or hypertext or a "Roguelike" front-end that produces a turn-based role playing game.

7.3 Final Remarks

In this thesis, I presented a system for procedurally generating general-purpose puzzles in video games. The system is composed of a puzzle generator library and two graphical tools that are flexible enough to be used for a wide range of games.

The puzzle generator library is implemented as a library for the popular Unity game engine and as such is ready for use by any developers who take an interest in this approach. To test the usability and flexibility of the system, I implemented a simple game (described in Chapter 6) that made use of all of the features of the system.

While this system has a few limitations and does not represent a perfect fit for all situations, I believe it is a viable approach to a largely unexplored problem in the fields of computer science and game design and can serve as a strong foundation for future work.

Appendix A

NP-Completeness Proof for Puzzle Generation

Claim: The problem of determining whether, given a puzzle map and database, the puzzle generator will be able to generate a puzzle is *NP-Complete*.

Proof:

To prove that the puzzle generation problem ($PGEN$) is NP-Complete, we must show that $PGEN \in NP$ and that $PGEN$ is NP-Hard. Proving that $PGEN \in NP$ is easy. Given a record of the inputs and outputs of each block in the puzzle map, we could verify in polynomial time that no rules or database properties were violated and that the record therefore corresponded to a valid solution. Since we can verify a $PGEN$ solution in polynomial time, $PGEN \in NP$.

To prove that $PGEN$ is NP-Hard, we reduce from $3SAT$. Given a boolean formula Ω of the form $C_1 \wedge C_2 \wedge ... \wedge C_n$ where each C_i corresponds to an OR clause with three variables (e.g. $< x_1 \vee \neg x_2 \vee x_3 >$), we construct a puzzle map and database such that the puzzle generator will be able to produce a puzzle if and only if Ω has a satisfying assignment.

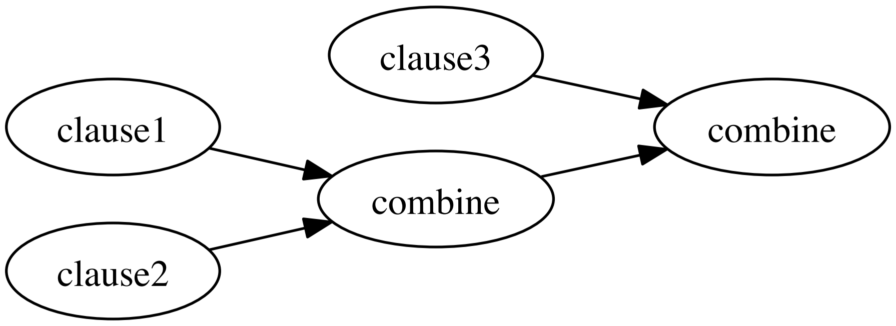


Figure A-1: Clause blocks cascaded across combine blocks to simulate “anding” the clauses.

First, each clause C_i is represented by a single puzzle block in our puzzle map and a single item in the database. These blocks are chained together by combine blocks as in Figure A-1 and additional items are added to the database to represent the results of this cascade of combines.

The result of this cascade is that the puzzle generator will only generate a puzzle if each of the clause blocks is able to successfully generate a puzzle. Now, to see how each clause block corresponds to a C_i in Ω, we examine each clause block in greater detail.

As shown in Figure A-2, each clause block is actually an OR block attached to three spawn inputs, each one spawning in its own area. Each of these areas corresponds to a specific value for a specific variable. Since area blocks can act as inputs to multiple blocks, these areas will potentially be attached to multiple clause blocks as necessary.

Now, since each of these variables corresponds to a specific value for a specific variable (e.g. $x_1 = true$), we must ensure that at most **one** of the areas corresponding to a specific variable x_i is spawned. It would make no sense if both the $x_1 = true$ and the

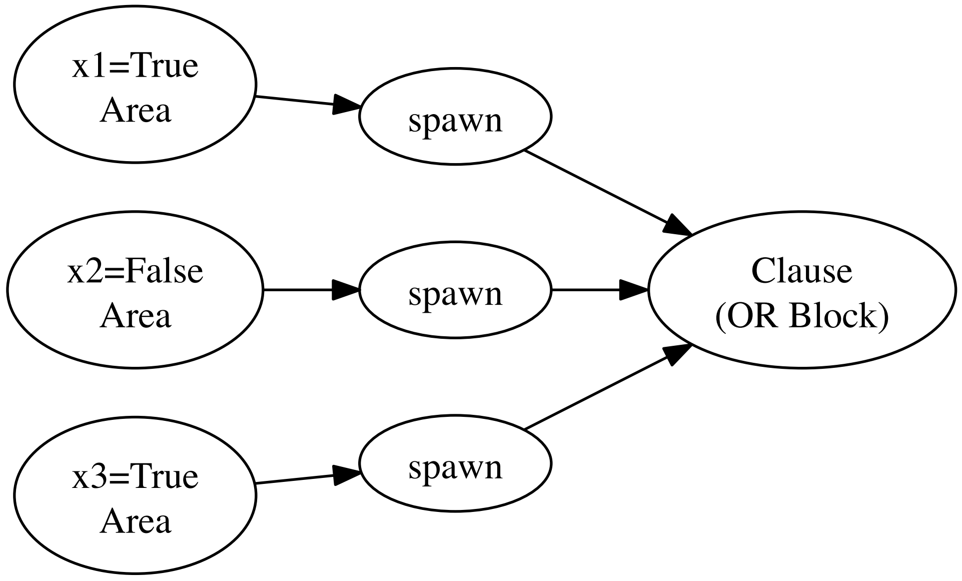


Figure A-2: Internal structure of a sample clause block. This one corresponds to $< x_1 \vee \neg x_2 \vee x_3 >$.

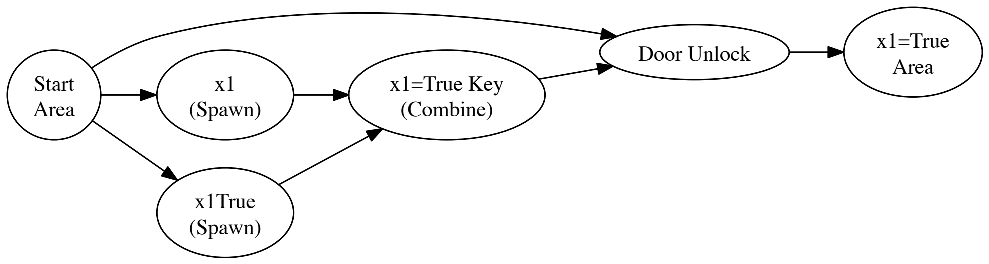


Figure A-3: The structure of an area corresponding to a specific variable and value. The spawning of x_1 ensures that only one of the areas corresponding to x_1 will be spawned.

$x_1 = false$ areas were spawned as this would correspond to the variable x_1 taking on both a true and false value.

To see how to prevent this, we look at each of these areas in greater detail. Figure A-3 expands the area blocks.

The area corresponding to $x_1 = true$ will attempt to spawn an item in the database called “x1” that is set to only be spawned once. Since the “x1” item can only be spawned once, we can be confident that at most one of the two areas $x_1 = true$ and $x_1 = false$ will be spawned. The additional door unlock, spawn, and combine blocks are included only because area blocks require either door unlock blocks or other areas as their input.

Now, since an area block spawning corresponds to a variable x_i being assigned a value of true or false and clause blocks will only successfully generate a puzzle if one of the variables in their clause is assigned a desired value, this constructed puzzle map and database will only be able to generate a valid puzzle if there exists a satisfying assignment for Ω, completing the proof that $PGEN$ is NP-Hard and therefore also NP-Complete.

Appendix B

Building Blocks Reference

This appendix includes a detailed specification of each of the building blocks built for the system at this time. Each section includes the expected input to a building block, the output produced by the block, the database properties used by the block, and an argument for why the block guarantees solvability.

B.1 Spawn Puzzle Block

The spawn puzzle block (structure in Figure B-1) is a base case that represents the simple act of the player picking up an item for use.

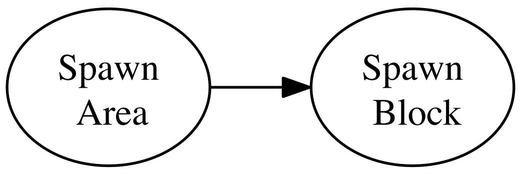


Figure B-1: Structure of a Spawn Puzzle Block

Property Name	Item	Format	Use
spawnArea	output item	text	Inserted by the spawn block to indicate the unique id of the area to spawn the output item.

Table B.1: Properties expected or used by the spawn block

B.1.1 Input

The expected input to a spawn block is an area block. This area block represents the spawn area for the item spawned by the spawn block.

B.1.2 Output

The output to the spawn block is the items and relationships necessary for the player to reach the spawn area along with the output item with the requested properties. The spawn block will also apply a "spawnArea" property to the item to indicate where it should be spawned. The Spawn puzzle returns no additional relationships.

B.1.3 Expected Properties

The properties expected and used by the spawn block are listed in Table B.1. In summary, the spawn block uses a property named "spawnArea" to indicate which area the output item should be spawned in.

B.1.4 Solvability

The spawn block guarantees that the player has access to the output item by spawning it in an area the player is guaranteed to have access to. The solvability guarantees of the spawn area block ensure that the player has access to any items spawned within it.

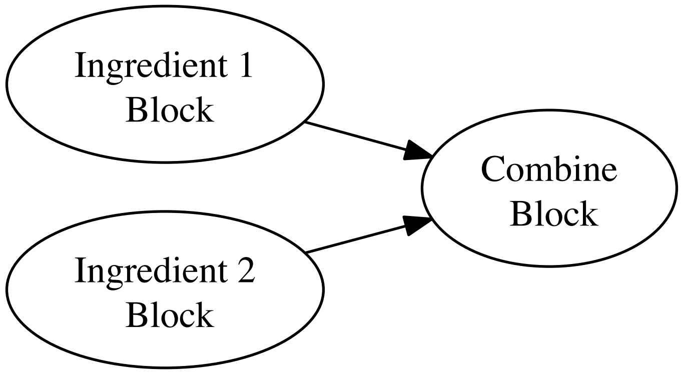


Figure B-2: Structure of a Combine Puzzle Block

B.2 Combine Puzzle Block

The combine puzzle block (structure in Figure B-2) is a building block that represents the act of the player combining two items together to produce a third item.

B.2.1 Input

The combine block requires two other puzzle blocks as its inputs. It attempts to generate the two "ingredients" used in the combination as the output to these input blocks. The combine block will also ensure that the two inputs can be transported to each other (either via containers or simple carryability).

B.2.2 Output

The combine block produces its output item via a combine relationship. Along with the items and relationships required to form the "ingredient" items, the combine block will return a combine relationship containing the output item.

Property Name	Item	Format	Use
madeby	output item	list of string pairs	Used by the combine block to determine which items can be used to construct the output item.

Table B.2: Properties expected or used by the combine block

B.2.3 Expected Properties

The properties expected and used by the combine block are listed in Table B.2. In summary, the combine block expects its output item to have the "madeby" property so it can determine which of the database items can be combined to produce this item. The "madeby" property is structured as a list of string pairs where each string pair is a pair of ingredients that can be used to form the item.

B.2.4 Solvability

The input blocks to the combine block guarantee that the player has access to the two "ingredient" items. Since the combine block ensures that the player can transport the two ingredients to the same location and that the two ingredients can be combine to form the output item, it guarantees that the player has access to the output item.

B.3 Item Request Puzzle Block

The item request puzzle block (structure in Figure B-3) is a building block that represents an NPC that requests an item from the player (sometimes with a specific property) and offers another item in return.

B.3.1 Input

The item request block requires two other puzzle blocks as its input blocks. One of these blocks represents the "requester" which requests an item. The other input block represents the "requested" item.

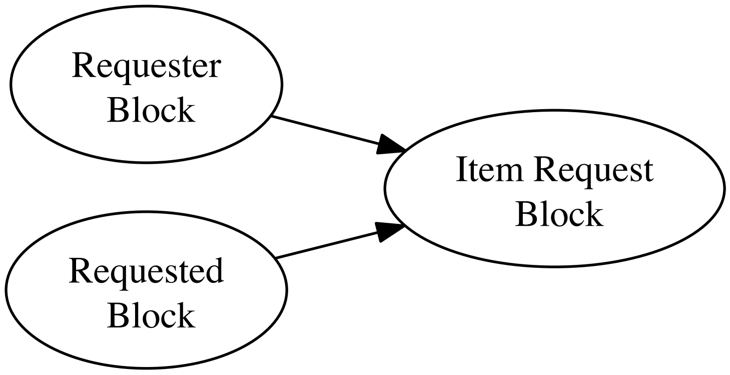


Figure B-3: Structure of an Item Request Puzzle Block

The item request block also ensures that the two inputs can be transported to each other (either via containers or simple carryability).

B.3.2 Output

The output of an item request block is the items and relationships necessary to generate the "requester" item and the "requested" item along with an item request relationship indicating that the requested item can be given to the requester to receive the output item as a reward.

B.3.3 Expected Properties

The properties expected and used by the item request block are listed in Table B.3. In summary, the item request block expects the output item to have the "givenby" property formatted as a list of items. It also expects the requester item to have the "requests" property formatted as another list of items.

Property Name	Item	Format	Use
givenby	output item	list of items	Used by the item request block to determine which items will provide the output item as a reward.
requests	requester input	list of items	Used by the item request block to determine what items a chosen requester might request.
mutables	requested input	list of properties	Used by the item request block to choose a property that the requester can require of the requested item. Useful for chaining item request blocks with property change blocks.

Table B.3: Properties expected or used by the item request block

B.3.4 Solvability

The input blocks to the item request block guarantee that the player has access to both the "requester" and "requested" items. Since the item request block ensures that the player can transport these items to the same location and that the requester will exchange the output item for the requested item, it guarantees that the player has access to the output item.

B.4 Property Change Puzzle Block

The property change puzzle block (structure in Figure B-4) is a building block that represents the act of a player using one item to change a physical property of another item. It should be noted that *A property change block by itself doesn't necessarily represent a puzzle required to complete the game.* When a property change block is provided as an input to another block that requires a certain property of its input (such as an item request block) then it actually becomes necessary for solving a puzzle.

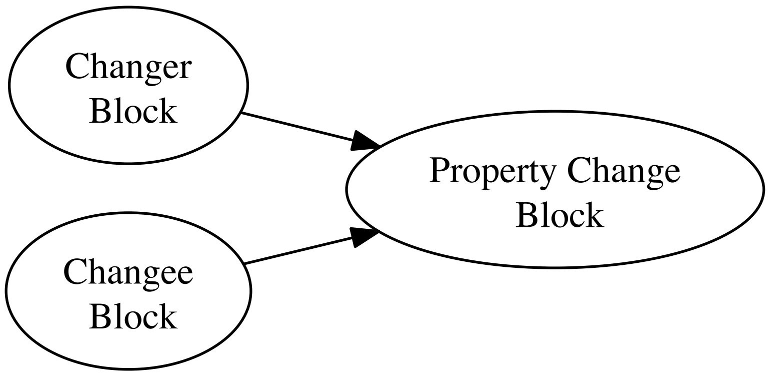


Figure B-4: Structure of a Property Change Puzzle Block

B.4.1 Input

The property change block requires two other puzzle blocks as its input blocks. One of these blocks represents the “changee” and is used to spawn the requested output item with an appropriate starting property. The second input block represents the “changer” and is used to spawn the item that will change the starting property of the changee to a requested (or randomly chosen) value.

The property change block also ensures that the two inputs can be transported to each other (either via containers or simple carryability).

B.4.2 Output

The output of a property change block is the items and relationships necessary to generate the “changee” item and the “changer” item along with a property change relationship indicating that the changer can change a certain property of the changee. The official output item produced by the property change block is the item spawned as the changee.

Property Name	Item	Format	Use
changedby	output item	list of items	Used by the property change block to determine the possible items to generate at the changer input.
mutables	output item	list of properties	Used by the property change block to determine what properties of the output item can actually be changed.
changes	changer input	dictionary mapping property names to a list of potential values	Used by the property change block to determine whether a possible changer item can actually change a relevant property of the output item.

Table B.4: Properties expected or used by the property change block

B.4.3 Expected Properties

The properties used and expected by the property change block are listed in Table B.4. In more detail, the property change block expects the output item to have the “changedby” property and one or more items present in a “mutables” property. Any potential changer item is required to have the “changes” property with the appropriate property names contained within. The expected format of the “changedby” property is a list of item names. The expected format of the “mutables” property is a list of property names. The expected format of the “changes” property is a dictionary mapping property names to a list of potential property values. Since the database editor as-is cannot produce the expected format of the “changes” property, a database extension is included that converts the format of the “changes” property from a list of string pairs to the expected format.

B.4.4 Solvability

The input blocks to the property change block guarantee that the player has access to both the “changer” and the “changee” items. Since the property change block

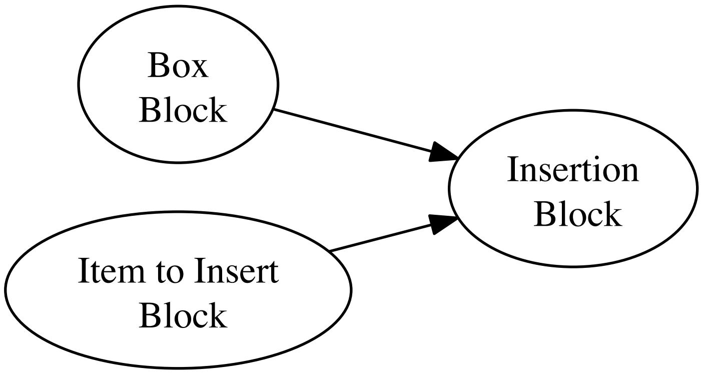


Figure B-5: Structure of an Insertion Puzzle Block

ensures that the player can transport these items to the same location and that the changer can change a relevant property of the changee, it guarantees that the player has access to the output item with the requested properties.

B.5 Insertion Puzzle Block

The insertion puzzle block (structure in Figure B-5) is a building block that represents the act of a player inserting one item into another. Similar to the property change blocks, an insertion block by itself is not necessarily a complete puzzle. It is very often combined with an unboxing block (Section B.6) to create a simple container puzzle that only becomes necessary for solving the entire puzzle if the item to insert into the container is not carryable by the player on its own.

In fact, since insertion blocks are almost always combined with unboxing blocks in practice, the simple puzzle map parser included with the current system will create a cascade of an insertion block followed by an unboxing block whenever it sees an insertion block in the output from the puzzle map generator. This structure is shown in Figure B-6.

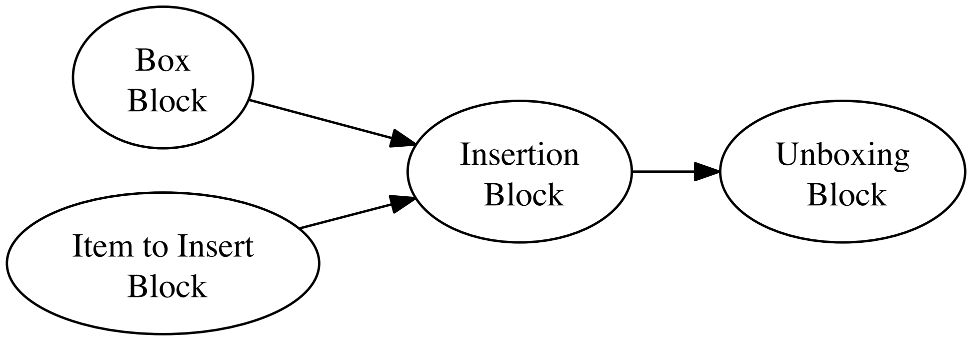


Figure B-6: Container block built by cascading an insertion block with an unboxing block

B.5.1 Input

The insertion block requires two other puzzle blocks as its input blocks. One of these blocks represents the “box” which can contain the other input. The other input block represents the “filler” which can be inserted into the box.

The insertion block also ensures that the two inputs can be transported to each other (either via containers or simple carryability).

B.5.2 Output

The output of an insertion block is the items and relationships necessary to generate the “box” item and the “filler” item along with an insertion relationship indicating that the filler can be inserted into the box. The official output item of an insertion block is the box item containing another item.

B.5.3 Expected Properties

The properties expected or used by the insertion block are listed in Table B.5. In summary, the insertion block expects the output item to have the “filledby” property formatted as a list of items. It also expects the output item to possibly have a

Property Name	Item	Format	Use
filledby	output item	list of items	Used by the insertion block to determine what items can possibly fill the output item.
contains	requested properties	list of items	Used by the insertion block to choose an item to generate at the filler input if a specific item is requested.
innerItemProps	requested properties	dictionary mapping property names to property values	Used by the insertion block to determine what properties to request at the filler input if a specific item is requested in the "contains" property.

Table B.5: Properties expected or used by the insertion block

requested property "contains" indicating the name of an item that should be generated at the "filler" input. Additionally, it expects the output item to possibly have a requested property "innerItemProps" formatted as a property dictionary indicating what properties should be requested at the "filler" input.

B.5.4 Solvability

The input blocks to the insertion block guarantee that the player has access to both the "box" and "filler" items. Since the insertion block ensures that the player can transport these items to the same location and that the filler can be inserted into the box, it guarantees that the player has access to the output item containing another item inside.

B.6 Unboxing Puzzle Block

The unboxing puzzle block (structure in Figure B-7) is a building block that acts as the inverse of the insertion block (Section B.5). It represents the act of a player *removing* an item from another item. As explained in Section B.5, an unboxing block by itself is not a complete puzzle and is often cascaded with an insertion block as in

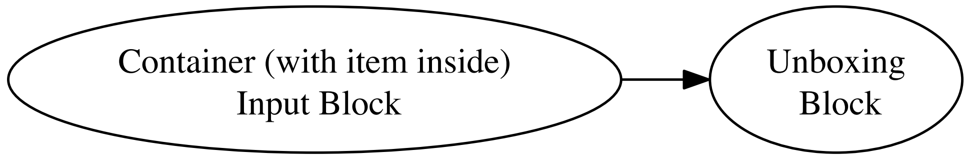


Figure B-7: Structure of an Unboxing Puzzle Block

Figure B-6 to produce a full container block.

B.6.1 Input

The unboxing block requires one puzzle block as its input. It will attempt to generate a "box" item containing its requested output as the output to this input block. Since it has only one input, it doesn't need to make any guarantees about the player transporting inputs to the same location.

B.6.2 Output

The output of an unboxing block is the items and relationships necessary to generate a "box" for the output item along with the output item itself. Although possibly redundant if cascaded with an insertion block, the unboxing block will also generate an insertion relationship to indicate that the output item can be placed inside some other item. The official output item of an unboxing puzzle is the requested item with requested properties and the guarantee that the output item can be removed from a container at some point.

B.6.3 Expected Properties

The properties expected and used by the unboxing block are listed in Table B.6. In summary, the unboxing block expectes the output item to have the "fills" property formatted as a list of other items. It uses the "contains" property and the "inner-

Property Name	Item	Format	Use
fills	output item	list of items	Used by the unboxing block to determine what items the output item can fit into.
contains	input item	list of items	Inserted by the unboxing block to inform its input that the box item should contain the output item.
innerItemProps	input item	dictionary mapping property names to property values	Inserted by the unboxing block to request specific properties for the item generated inside the box at the input block.

Table B.6: Properties expected or used by the unboxing block

ItemProps" property to communicate that the output item is inside another item to its input blocks.

B.6.4 Solvability

The input blocks to the unboxing block guarantee that the player has access to a "box" containing the requested output item with requested properties. Since the unboxing puzzle merely ensures that this input is generated and that the output item can be removed from the box, it guarantees that the player has access to the output item.

B.7 Door Unlock Puzzle Block

The door unlock puzzle block (structure in Figure B-8) is a building block that represents a locked door between two areas. It generates a "key" item used to unlock the door.

B.7.1 Input

The door unlock block requires two input blocks. One of the blocks is an area block representing the "source" area of the locked door (i.e. the side of the door that

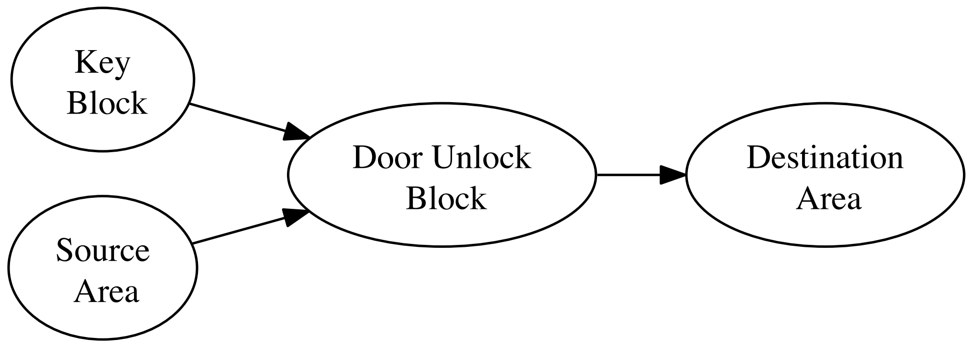


Figure B-8: Structure of a Door Unlock Puzzle Block

the player can reach without the key). The other input block is a puzzle block representing the "key" used to unlock the door. To generate a unique key for the areas in question, the door unlock block will attempt to create a new database item by copying an existing "key" entry in the database.

The door unlock block also ensures that the key can be transported (either in a container or by hand).

B.7.2 Output

Door unlock blocks can only act as input to an area block. As such, a door unlock block does not expect to generate an output item. Instead, it expects to know the ID of the "destination" area block so that it can generate an appropriate locked area connection relationship along with all the items and relationships necessary to generate the "key" item and provide access to the "source" area.

B.7.3 Expected Properties

The door unlock block does not expect any properties of its items. Instead, it expects an item named "key" to exist in the database that it can use to generate a unique key.

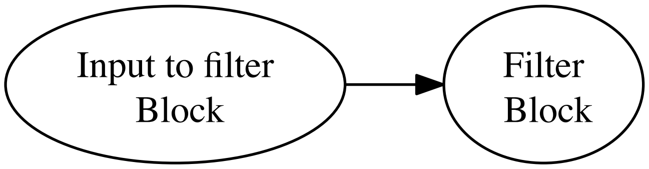


Figure B-9: Structure of a Filter Block

B.7.4 Solvability

The inputs to the door unlock block guarantee that the player has access to the "source" area of the locked door along with a "key" used to unlock the door. Therefore, the door unlock block guarantees that the player has access to the "destination" area that maintains the door unlock block as its input.

B.8 Filter Block

The filter block (structure in Figure B-9) is a building block that can be used to filter out certain inputs from being considered. When a filter is created, it is given a list of database properties and required values. A required value of "None" indicates that, to pass the filter, a given item must possess *some* value for the given property.

B.8.1 Input

The filter block requires one puzzle block as its input. If the requested output item passes its filters, it will attempt to generate the output item at its sole input. Since it has only one input, it doesn't need to make any guarantees about the player transporting inputs.

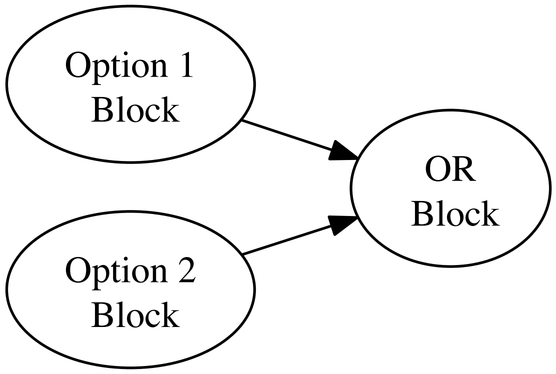


Figure B-10: Structure of an OR Block

B.8.2 Output

The output of a filter block is the items and relationships necessary to generate the output item, assuming the output item passes its filter.

B.8.3 Expected Properties

The filter does not expect the output item to have any specific properties.

B.8.4 Solvability

Filter blocks are unrelated to solvability. They are primarily a tool that allows the designer greater control over the puzzles produced by the generator. One possible use of filter blocks is to force the generator to choose a certain solution by altering the puzzle map. Filters can effectively be used to remove non-determinism from the generation process.

B.9 OR Block

The OR puzzle block (structure in Figure B-10) is a building block that randomly chooses what order to forward its output item to its inputs, allowing for a more dynamic puzzle map.

B.9.1 Input

The OR block requires two puzzle blocks as its inputs. It will attempt to generate its output item at both of these blocks in a random order, using the first input that successfully produces the output item.

In terms of transporting items, the OR block will only report that its output can be transported to another location if *both* of its inputs report that the item is portable (either via container or simple carryability). The reason for this is that if the output item is carryable when spawned at input 1, but not at input 2, the OR block can't guarantee that it *will* spawn the output item at input 1, so it has no choice but to be conservative.

B.9.2 Output

The output of an OR block is the items and relationships necessary to generate the output item, provided by one of the input blocks.

B.9.3 Expected Properties

The OR block does not expect the output item to have any specific properties.

B.9.4 Solvability

One of the input blocks to the OR block guarantees that the player has access to the output item. Additionally, the OR block will tell its output block that its output item can be transported only if the output item would be carryable when spawned at

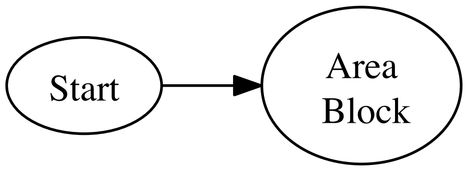


Figure B-11: Structure of start area block

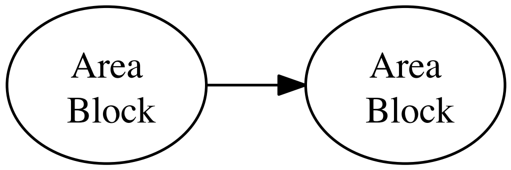


Figure B-12: Structure of an area to area connection

both of its input blocks, ensuring that the output block of an OR block won't make any mistakes in determining carryability.

B.10 Area Block

The area block is a building block that represents a single unit of location within a game. It might represent a single room or an entire floor. The sample game outlined in Chapter 6 implements areas as a set of connected rooms. Area blocks are different than other building blocks primarily because they accept multiple output requests. For instance, multiple spawn blocks might spawn an item in the same area. Area blocks also have more restrictions on their inputs and outputs.

B.10.1 Input

There are only three possible inputs to an area block. An area block might have one of the following as its input:

1. A special “Start” block indicating that this area block represents the area in which the player starts the game. See Figure B-11.

2. Another area block, indicating that this area is connected to another area. See Figure B-12.

3. A door unlock block, indicating that this area is connected to another area via a locked door. See Figure B-8.

Regardless of what its input is, an Area will try to generate *anything* at its input. If its input returns successfully, that indicates that the player is guaranteed to be able to reach this area, so the area itself can then return a success message.

B.10.2 Output

Again, there are only a few blocks that can include an area block as their inputs. Spawn blocks, door unlock blocks, and other area blocks are the only blocks that can attempt to generate an area. As output, an area block will return all items and relationships necessary to reach that area. Additionally, the area block will return an area connection relationship if necessary to connect it to its output block.

Important note: Because an area allows multiple blocks to include it as an input, it has to be very careful about when to deallocate any of its inputs. Right now, the first block to attempt to generate an area block becomes “bound” to the block and receives the appropriate output from the area. Subsequently, any blocks that attempt to generate the same area block will receive a success message but an empty list of items and relationships to spawn. Subsequently, the area will only deallocate its inputs if the block “bound” to it tells it to.

B.10.3 Expected Properties

An area block does not expect the database items to have any properties.

B.10.4 Solvability

The input blocks to an area guarantee that the area is reachable from the player's start location. Since the area then generates the relevant connection relationships to ensure that any output areas are also reachable from it, it enforces the same guarantee on its outputs. Since a successfully generated area is guaranteed to be reachable from the player's start location, any items spawned in the area are guaranteed to be accessible.

B.11 Composing Building Blocks

Each building block is designed to be completely abstracted, so that the composition of any number of building blocks will appear to act and behave like a single block.

One common example of composing multiple blocks to act like one block is the common pattern of cascading an insertion block and an unboxing block as shown in Figure B-6. As was mentioned in Section B.5, even this cascaded block does not always produce relationships actually necessary for solving a puzzle. For instance, the cascaded block will generate a "box" for the output item, but if the output item is carryable by the player, then the generated box is not necessary to transport the output item, meaning the system generated a superfluous item.

If the designer wanted to generate a more strict "Container" puzzle that actually ensured that the generated box was required for the puzzle's solution, she might choose to create another composed building block like the one in Figure B-13. This composed block uses filters to ensure that neither input to a combine block is carryable by the player and therefore a container must be used to transport the ingredients to the same location.

While this "Combine Container" block in Figure B-13 looks complex, once constructed, it can be thought of as just another building block as in Figure B-14. Com-

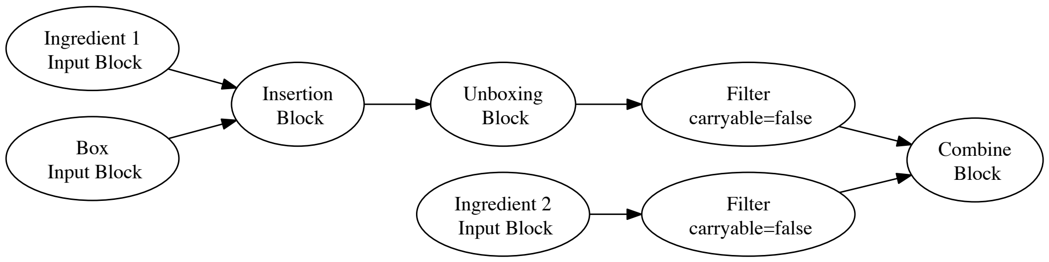


Figure B-13: A full “Container” puzzle block built out of existing building blocks

posing building blocks in this way improves generality and grants the designer greater control over the generated puzzles.

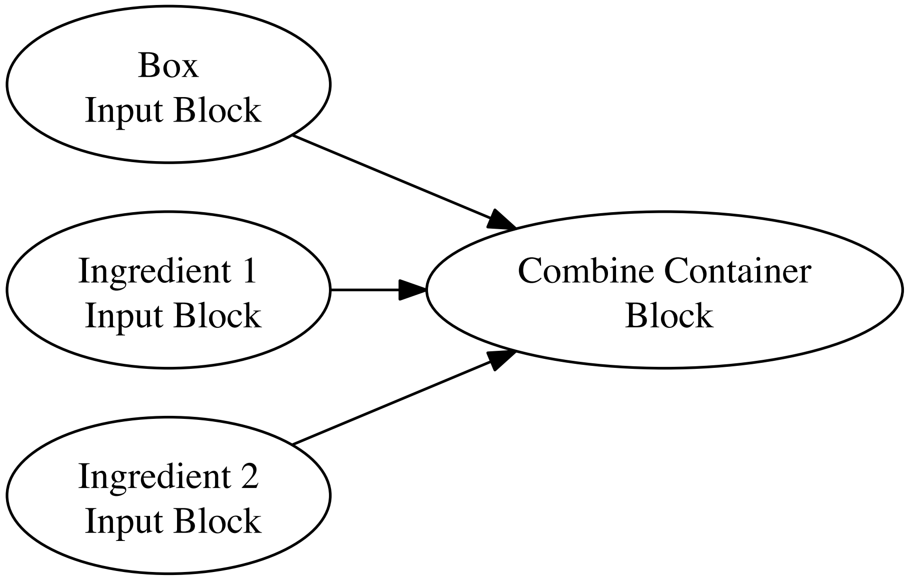


Figure B-14: A simpler representation of the composed block in Figure B-13

Bibliography

[1] Tarn Adams and Zach Adams. Slaves to armok: God of blood chapter ii: Dwarf fortress. [Online], 2006.

[2] Calvin Ashmore. Key and lock puzzles in procedural gameplay. Master's thesis, Georgia Institute of Technology, April 2006.

[3] Kate Compton and Michael Mateas. Procedural level design for platform games. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)*, June 2006.

[4] Jonathon Doran and Ian Parberry. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[5] Joris Dormans. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 2:1–2:8, New York, NY, USA, 2011. ACM.

[6] Singapore-MIT GAMBIT Game Lab. Symon. [Online], 2010.

[7] Singapore-MIT GAMBIT Game Lab. Stranded in singapore. [Online], 2011.

[8] Edmund McMillen and Florian Himsl. The binding of isaac, 2011.

[9] Microsoft. Minesweeper, 1990.

[10] Mojang. Minecraft, 2011.

[11] `http://upload.wikimedia.org/wikipedia/en/1/17/Rogue_Screen_Shot_CAR.PNG`. Accessed April 16, 2013.

[12] `http://spelunkyworld.com/images/spelunky-pc-screen.png`. Accessed April 16, 2013.

[13] `http://gambit.mit.edu/images/stranded4.jpg`. Accessed April 20, 2013.

[14] `http://gambit.mit.edu/images/symon2.jpg`. Accessed April 20, 2013.

[15] Michael Toy, Glenn Wichman, Ken Arnold, and Jon Lane. Rogue, 1980.

[16] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *CoRR*, abs/1201.4995, 2012.

[17] Derek Yu. Spelunky. [Online], 2009.