Creative Coding - Week 6

Game Design with Puzzlescript

Lecturer: Dr Michael Cook (mike.cook@kcl.ac.uk)

Figure 1: A screenshot of *Skipping Stones to Lonely Homes*, by Alan Hazelden (made in PuzzleScript).

This week is the first of three weeks looking at game design. Game design is a huge discipline that covers many different problems, styles, topics, skills and creative goals. We could easily fill a four-year course discussing how to make and think about making different kinds of games, at different scales. The next three sessions will focus on the following things:

- Teaching you how to use small, efficient and popular tools for game prototyping.
- Focusing on how to make small games that you can finish and share with others.
- Thinking about design problems and working in multiple creative areas at once.

The best thing you can do if you want to make games, for any reason, is to **make (and finish) a lot of very small games** and **put them on the Internet**. This is a lot harder to do than it sounds, but the good news is that the difficulty is mainly psychological rather than technical. I'm still practicing it after fifteen years!

This week we'll be looking at PuzzleScript, a browser-based tool for making puzzle games. PuzzleScript was designed by Stephen Lavelle, also known as *increpare*, a prolific indie develop who has made hundreds of games over the last fifteen years. PuzzleScript is accessible and easy to use, and can quickly whip up puzzle prototypes once you know how to use it. It's been used to create thousands of puzzle games, as well as prototype versions of larger commercial games like Cosmic Express or A Monster's Expedition.

You can see a selection of PuzzleScript games in the gallery here, if you want to sample some before we start. PuzzleScript has intentional limitations that restrict what you can and can't do. This can take some getting use to but is one of the reasons it's so good as a creative tool!

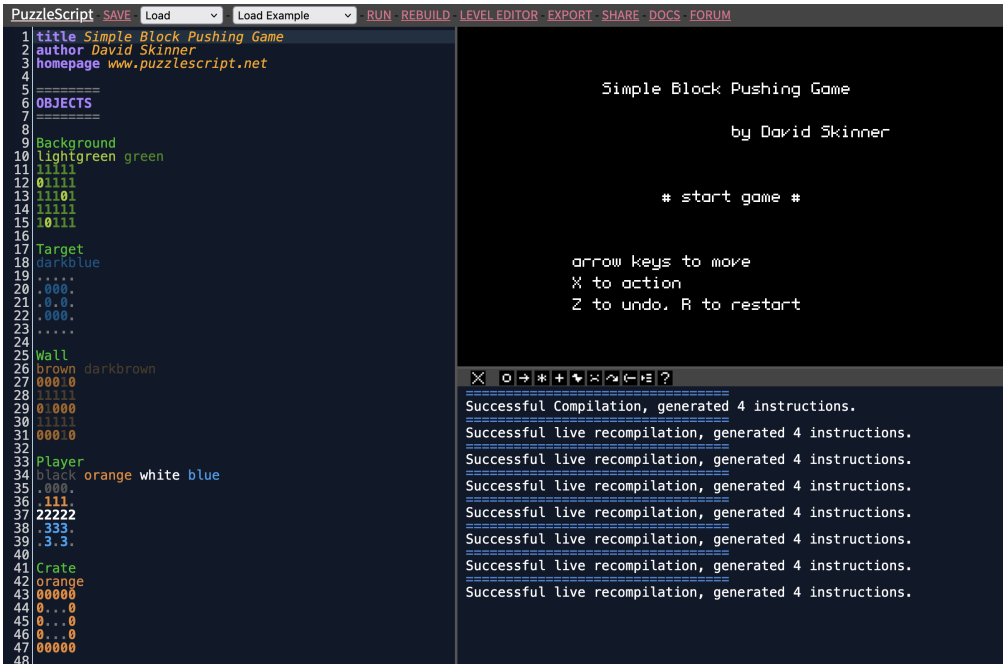

Figure 2: The PuzzleScript editor when first opened.

1 A Tour of PuzzleScript

Today's session has two parts: a playable tutorial I've made, and then an open-ended design problem. First, let's take a quick tour of PuzzleScript itself. PuzzleScript has its own tutorial guide if you need it too, and of course I'm always on-hand to answer questions.

Click here to open the PuzzleScript editor page.

The PuzzleScript editor has three panes by default. On the left is your source code - this contains the entire game, including sprite designs, rules, objectives, level designs and text. In the top-right is the game window. This is where you'll test your game, and also what players will see when they play it. In the bottom right is the log, for errors and other debug output. If you click on the top-right pane where the game window is and press X, the current game will run and you can play it (use the arrow keys to move and X to interact).

1.1 PuzzleScript Game Files

Let's focus on the code, the left-hand pane. A PuzzleScript game has several sections, you can scroll down with me as I go. I'm going to be using *Simple Block Pushing Game* as a reference, you can find it by clicking 'Load Example' at the top of the page, and selecting 'Basic'.

1.1.1 Objects

Objects in PuzzleScript are 5x5 pixel sprites. You define them first by giving them a name (like 'Wall' or 'Crate'), then defining a palette of colours, and finally by drawing a grid of numbers. The

```
# # # # . .
# . O # . .
# . . # # #
# @ P . . #
# . . * . #
# . . # # #
# # # # . .
```

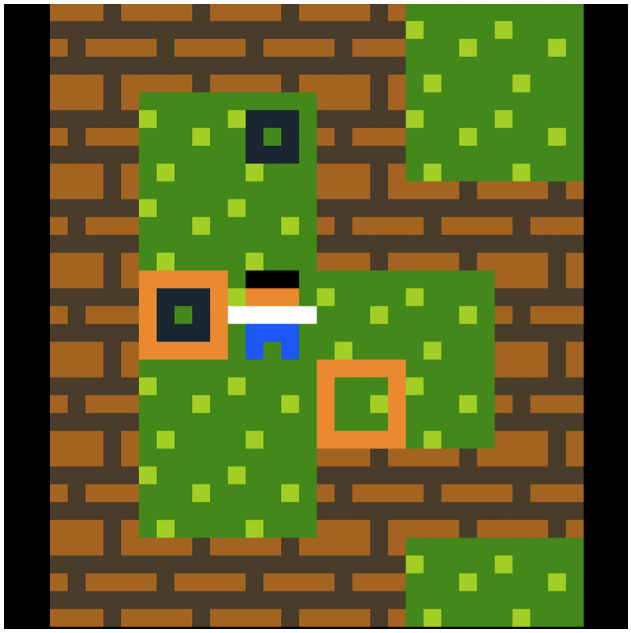

Figure 3: An example PuzzleScript level definition, next to its rendering in the game.

numbers in the grid index the palette you defined, so if 'darkblue' is the first colour in your palette, then every 0 in your sprite will become a dark blue pixel. If 'red' is the second colour, every 1 will become a red pixel, and so on.

Here's a full list of colours:

```
black white lightgrey grey darkgrey red darkred lightred brown darkbrown
    lightbrown orange yellow green darkgreen lightgreen blue lightblue darkblue
    purple pink transparent
```

1.1.2 Legend

In your level designs later you'll use a grid of symbols to tell PuzzleScript where to put objects in the game levels. To do this we need to create a *legend* which tells PuzzleScript how we're going to represent each object in the level maps. In the example game, we can see the `#` is used to represent a wall object, for instance. To see this in action, let's take a detour down to the bottom of the file...

1.1.3 Levels

In this section we define the levels of our game. PuzzleScript will show players the first level when they start, and then every time the win condition is met (which we'll get to in a bit) it will take the player to the next level in the sequence, until the game is finished. Figure 3 shows the code for Level 1 of the example game next to a screenshot of the level. You can see how each symbol defined in the Legend matches up with something in the level.

There's something special about this example. The symbol for Crate is `*` and the symbol for Target is `O`, but our level starts with one of the crates already on a target. We can't put two symbols

into the same spot in a PuzzleScript level design, so to do this we can define a new symbol that captures this state. If you go back to the Legend you'll see there's a symbol defined as follows:

```
@ = Crate and Target
```

This tells PuzzleScript that when it sees an `@` in a level design, it should add both of those objects there. This is really important for certain kinds of complex level design.

PuzzleScript has a few advanced features in its level designs, but we're going to leave them for now. To read about them, check the *Advanced PuzzleScript* section later in this document. Ok, let's scroll back up to Collision Layers.

1.1.4 Collision Layers

One of the most fundamental things that happens to objects in PuzzleScript is that they move. We'll write the rules for movement in a bit. However, some objects should block each other from moving - for example, a wall should stop the player. In game development we call this 'collision', and PuzzleScript has a section for defining which objects are on the same 'collision layer'. Any objects on the same collision layer will stop each other from moving, and any objects on different layers will ignore each other.

1.1.5 Rules

Rules are where the logic of your game is defined. PuzzleScript uses a *very compact* rules language that is also quite arcane and hard to parse at first. The block pushing game has just one rule:

```
[ > Player | Crate ] -> [ > Player | > Crate ]
```

Let's look at this bit by bit. PuzzleScript's rules are pattern replacements: it looks for the pattern on the left, and then replaces it with the pattern on the right. The left-hand side of this pattern says "the player is moving into a crate". There are a few parts to this.

`[ X | Y]` is a pattern in PuzzleScript to describe two tiles that are next to each other. This says that X and Y are next to each other in the game world. We can write longer rules like `[X | | Y]` to describe X and Y having an empty tile between them, or `[X | Y | X]` to describe an X being flanked by two Ys.

The `>` symbol means an object is moving. This might be because another rule caused the object to move or, in this case, because the object is the player and the keyboard is making it move. The direction is also important. This says that the player is moving towards the crate – because the arrow is pointing towards the tile containing the crate.

The right-hand side of this rule says 'the player and the crate are moving in the same direction'. Notice that there is now a `>` symbol next to the crate too, and that it is pointing in the same direction as the player. This says that if the player is moving left and bumps into a crate, the crate will move left as well.

PuzzleScript automatically rotates rules like this to apply in all four directions. The `>` symbol only matters relative to the rule layout here, but in the game it applies to pushing a crate up, down or left as well. If we flip the symbol around, it means something different:

```
[ < Player | Crate ]
```

This says 'the player is moving away from the crate'. Whether they are moving left or right doesn't matter - the rule is defining movement in relation to the crate.

For more information about the rules language, check out the tutorial here. Another good way to learn is to look at the code of a PuzzleScript game, since every one is open source!

1.1.6 Win Conditions

Finally, we need to say what the player is trying to achieve. When this condition is met, PuzzleScript will load the next level. You can have as many win conditions as you want, but they must *all* be true before a puzzle completes.

Most win conditions say something about the state of the level, and are fairly self-explanatory. Here are examples from the official tutorial, with more context from me:

```
(no objects called Fruit are in the level)
No Fruit

(all objects called Target are on a space that also contains a Crate)
All Target On Crate

(at least one object called Love exists in the level)
Some Love

(at least one object called Gold is on an object called Chest)
Some Gold on Chest

(no objects called Gold are on the object called Chest)
No Gold on Chest
```

Figure 4: A thrilling level from *Escape from Bush House*

2 Task 1: Escape From Bush House

You now have two options for today! If you like, you can skip these exercises and start working on a game of your own. There are lots of examples to look at for inspiration, or you can read the tutorials or other articles. Feel free to go on from here and work on something yourself. Otherwise, read on for two tasks you can do to try out specific things with PuzzleScript.

For the first task, I've designed a game for you that'll also give you tasks to complete to practice PuzzleScript! You can find it here:

Escape from Bush House

Each level has something wrong with it, which you can fix by editing the game code. **After you change the code, click 'Run' at the top of the screen to rerun the game.** It should remember which level you were on. If you get stuck, I've uploaded a version of the game with all the solutions commented. You can find the solutions here.

3 Task 2: Progression Design

Some games have tutorials that tell the player exactly how to play them. However, puzzle games often do away with this and instead teach the player through the design of their puzzles. This is very common in PuzzleScript games. I personally believe this trend has become overused in game design, but it's still an important skill that is worth practicing!

For this task, you're going to take an existing game and design a set of levels for it. Each level is going to teach the player about a new concept. When you design each level, think about:

- What has the player already demonstrated they know?
- What do I want to teach the player next?
- What assumptions will the player make and how can I use that to teach something?

Figure 5: A screenshot showing a test level of Magnatron (and a message from my finished version of the game, linked below).

3.1 Teaching the Basics

The game you're going to design levels for is called **Magnaban**. It's a small game I made for this week's session, adapted from Alan Hazelden's *Sticky Candy Puzzle Saga* and the basic block pushing game we looked at earlier. The rules are the same as the block pushing game (sometimes known as *Sokoban*) but with one extra rule: if two boxes touch each other, they stick together forever.

Click here to access the template code for Magnaban.

First, let's list out the basic rules that we want the player to learn:

1. The arrow keys move the player.
2. You can push boxes by moving into them.
3. Pushing all the boxes onto targets wins the level.
4. Boxes stick together once they touch, but you can still push them.

These are the basic 'facts' of the game that we want to teach. We want to figure out how to gradually introduce these ideas to people. I recommend designing a level that teaches points 1-3 together, then a second level that teaches point 4. For each level, think about the minimal level you need to design that teaches the player these concepts. **Take a moment to design two levels to teach these two ideas now.**

3.2 Teaching Concepts

After we teach the players the basics, we can think about what interesting *consequences* these rules have. In puzzle games these often take advantage of assumptions that players have made, or subtle effects they may not have noticed. Here is a list of further concepts we might want to teach them:

- Sometimes, sticking two blocks together is bad (you can't separate them to put them on targets that are far apart).
- Sometimes, sticking two blocks together is useful (you can push one block and they will all move together, letting you shuffle blocks into tight spaces).

These two examples play off each other. To teach the first one, we might design a level where you have to carefully guide two blocks down a narrow corridor without them touching each other. Then in the level after, we might design a level where the only way to solve the level is to stick the blocks together. These test player assumptions and also guide them to learning new concepts. **Try designing two levels to teach these concepts now.** If you need inspiration, try playing Alan Hazelden's Sticky Candy Puzzle Saga. The first level teaches a lot of things all at once (Alan's games are extremely compact and do not waste time!)

3.3 Adding New Ideas

Puzzle games typically have a language of rules that the player learns the dynamics of. For example, first they learn they can push boxes, then they learn boxes stick to each other, but they may also learn other subtler effects like the fact that pushing a box into a corner means it can't be recovered. Over time each of these little ideas become things they can combine together to solve the problems you design.

When we introduce a new concept we not only add new rules to the system, but we also add potentially dozens of new combinations of rules as well. The new concept is itself a set of new rules, but it might also combine with existing concepts to create new things that you can do, and let you solve problems that were impossible beforehand. There's one extra game concept I've added to Magnaban: an inverse magnet box which pushes magnets away from it.

If you look at the second level of my examples, you'll see the effect of the inverse boxes in action. If you push them towards regular crates, the regular crates get pushed away. For your final task, **think of a consequence of this new mechanic** and then **design a level to teach it**. Remember to introduce things slowly - you need to teach the player what the inverse magnet block does before you can ask them to solve a problem with it, so you may want to break that into two separate levels.

There's no perfect answer to any of these things, but if you want to see what I came up with, my level designs can be played here:

Magnaban (Mike's Edition)

Level design, game design, all of these things are interesting and complicated creative skills that take a long time to practice. Well done for getting this far today!

4 What Next

4.1 Share your work with me!

I'd love to see what you make, and I'd also love to be able to share people's work at future department events, or show to students in future years for inspiration! I would love it if you shared what you make - anonymously if you prefer, or with your name for full credit.

Ways you can share your work:

- Email the source code or a link to your PuzzleScript game to my email directly (mike.cook@kcl.ac.uk)
- Post it to GitHub and share with me (username: gamesbyangelina)
- Drop it anonymously in this form: https://forms.gle/qh8UWnqW1fNJyfxz8.

PuzzleScript has an 'Export' button which will download your game as an HTML file, and a 'Share' button which can automatically host it on GitHub (once set up). It's very easy to share games this way!

4.2 Further Reading

Stephen Lavelle, the inventor of PuzzleScript, has made several amazing puzzle games himself. The most infamous is perhaps Stephen's Sausage Roll, a very hard puzzle game that is packed full of secrets. He has also made other game-making tools, like flickgame which is a tool for making little games out of hyperlinked images. He has recently been posting about what appears to be a new tool for making little 3D games that I'm very excited about.

The best way to understand how puzzle games are designed is to play a bunch of them. The PuzzleScript gallery has a lot on there already, but if you want to play some bigger commercial games I recommend ones designed by Alan Hazelden (developed by Draknek & Friends). They show this elegant progression of teaching ideas slowly and learning how to combine ideas together really well. For an even bigger, even more commercial example, going back in time and playing Portal or Portal 2 is also a good way of seeing how a major game studio approaches the same problem. It's worth noting that there are many different styles of designing puzzles and levels though, and many different ways to teach the player how to do things. Some games are more open-ended, some games offer branching paths with different solutions. There's no 'correct' way to design games.

If you want to share games you've made, I recommend signing up for an account on itch.io. itch.io is a place where people can sell or just host their games for free, including downloadable and browser-based games. It has a great community and is host to a lot of game jams and other community events. I have a page on there too! You can find my games here.