PAC-MAN EXTREMEMEEEEEEEE!!!!!™

Kim Dauber and Rachael Devlin

Abstract

Is regular old Pac-Man too boring for you? Your boring days of dot-gobbling are over, because here comes PAC-MAN EXTREMEMEEEEEEEE!!!!!™

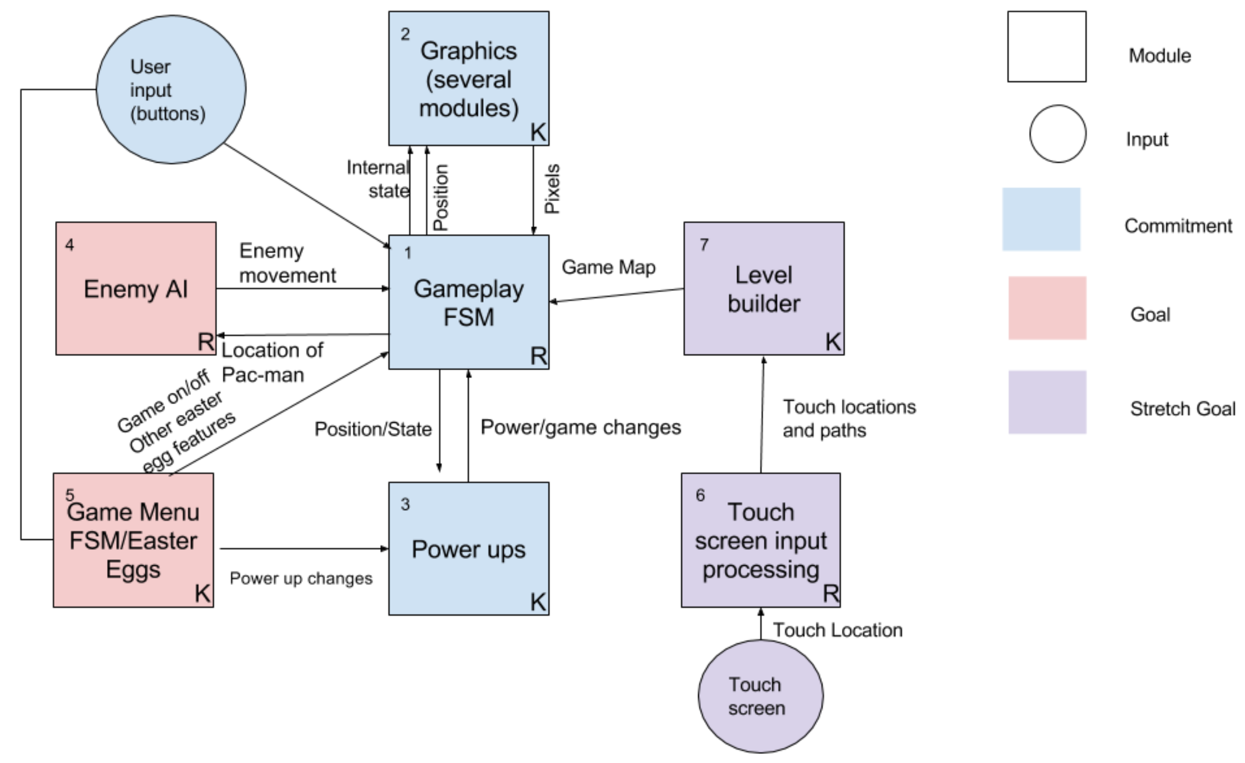
PAC-MAN EXTREMEMEEEEEEEE!!!!!™ is two-player Pac-Man with powerups and a level editor. Grab powerups to destroy your competitors' chances at victory. Use a touchpad to create a level of Pac-Man. You draw the walls and the FPGA will make your vision come to life, ready to play.

Introduction

The base modules of this project are the graphics modules and the gameplay finite state machine. These modules will integrate the others in order to make the game appear on the screen and to track the player's state in the game.

We intend to use buttons on the FPGA and Nintendo NES controllers for user input. However, we will need to use a touch screen for level building. These generally cost $10-$20 maximum, but we have been told that we can borrow one from the class.

Block Diagram



Modules

Commitments

Graphics

The graphics modules provide VGA input to the gameplay and game menu finite state machines in order to render actual graphics, rather than simple blocks, to the game. Our plan is to have the game screen's total size be 600 x 600 pixels. This would have a single game board being 600 x 300 pixels, putting one game on the screen for each of the two players. We will need to store at least the following sprites to potentially displayed on the screen:

- Pac-Man (12-18 variations, depending on the direction he's facing and how open his mouth is)
- Dot (for gobbling - two variations, either eaten or blank)
- Enemy (possibly in two variations for an animated effect)
- Powerups (arbitrary variations depending on the number of powerups)
- Walls (one to eleven variations, depending on whether a wall is a solid block of color or real lines)
- Letters and digits (for the game menu and scoreboard)

Each sprite will be 16 x 16 pixels and will have all of its pixels information stored in memory. The minimum number of sprites required is 16, and we estimate the maximum to be 40. Using 64 colors, we expect graphics to use at minimum
$16\ sprites\ \times (20\ \times 20)\ pixels\ \times 6\ color\ bits\ =\ 4.8\ kB$ and at maximum
$80\ sprites\ \times (20\ \times 20)\ pixels\ \times 6\ color\ bits\ =\ 24\ kB$ at maximum. This will fit into the Nexys 4's 12 MB of internal RAM. Storing each bit directly and fixing the size of each sprite at 16 x 16 pixels will reduce the complexity by removing some of the arithmetic operations required to display sprites with rounded edges.

The inputs to the graphics modules are the positions (from the gameplay FSM) and the internal state (such as the direction Pac-Man is facing and how much his mouth is open). The output is one pixel of the screen, with timing synchronized with the monitor's display.

Creating a sprite literally in verilog requires a great deal of code, so in order to facilitate this we will manage our sprite graphics through a web app called "Sprite Maker," located at http://kadauber.scripts.mit.edu/sprite-maker/. We built this app ourselves.

Gameplay Finite State Machine

The gameplay finite state machine keeps track of the locations of all the sprites and performs logic regarding their interactions on the screen. For instance, it can tell whether Pac-Man can move into a wall, how many points each player has earned, or whether an easter egg has been activated. The implementation will mainly be a major FSM composed of minor FSMs, such as an FSM for the enemies. The greatest complexity will be ensuring that all the functionality works properly. Therefore, we will manually test each piece of the gameplay FSM's functionality as we add new states. We can use this strategy because Pac-Man progresses at speeds humans can interact with.

There should be one gameplay FSM for each player (so two for two player). For two player games the gameplay FSM will have inputs and output to connect the two so game over and powerups affect both players as necessary.

Inputs

- Pixels from the graphics
- Game map from the level builder
- Power/game changes from the powerups
- Game on/off and other easter egg features from game menu and easter eggs
- Enemy movement from the enemy AI
- User input from the buttons

Outputs

- The positions and states of all the graphics
- The positions and states of all the powerups for the powerups
- The positions and states of the enemies for the enemy AI

Powerups

We have not yet chosen specific powerups for this game yet; however, they will definitely include the ability to eat ghosts, since that was in the original Pac-Man. The next most likely one is being able to become a ghost in your opponent's game and take one of their lives by running into them. Others could be moving at twice your usual speed, freezing your opponent's game, or making all the dots on your opponent's board disappear (without this allowing them to win, so they must run from ghosts but cannot progress in the game). The powerups' states will act as an input to the major gameplay FSM. The gameplay FSM will provide an input to the powerups module instructing it to enable or disable the powerups it has. Complexity for the powerups will mainly have to do with their interaction between the two gameplay FSMs, their interactions with other active powerups, and their timing.

Powerups input the positions and states of the powersup from the gameplay FSM. They output the power and game changes to the gameplay FSM.

Goals

Enemy AI

The enemies in Pac-Man originally move around the board in Pac-Man's general direction. This is a relatively simple rule-based artificial intelligence that would take as inputs the locations of Pac-Man and the enemies and would output instructions for the direction the enemies should go next. This is complex in the need to test the logic so that the enemies do not get stuck in a single location, are not too hard to beat, and are not too easy to beat. It may also need to interface with powerups depending on which powerups we choose.

Inputs to the enemy AI are the position of Pac-Man and the positions and states of the enemies from the gameplay FSM. Outputs are the enemies' movements in response to those inputs.

Game Menu + Easter Eggs

The game menu is a large FSM that will allow the user to enter game mode or the level builder or other easter eggs. It will have a simple graphical interface and require few computations and most of the memory need is for displaying letters which is accounted for in the graphics module. The Easter Eggs will most likely affect certain powerups and graphics and can only be activated in the game menu. The complexity in the easter egg lies in interfacing with the larger gameplay FSM. This can be mitigated by limiting the effect of the easter egg and extensive testing for bugs during integration. Obviously, we cannot specify what the easter eggs specifically are because the nature of easter eggs is to be secret.

The game menu takes only user inputs. Its outputs are the game states and easter eggs' effects, which will definitely go to the gameplay FSM and may also go to powerups if the easter eggs affect the powerups.

Stretch Goals

Level Builder

A level builder is our main stretch goal for this project. It allows players to create levels not originally included in the game. Players can do this by using the touchscreen to place walls on the gameboard. (The ghosts' original location would not move. Any space not occupied by a wall would have a dot that Pac-Man can gobble.) The level builder would require at least 60 bytes of memory (one bit for each of the 450 available spots on the board). The level builders main complexity lies in dealing with ambiguous inputs. The first was to deal with this is to ignore them. However, if this makes the input too difficult to use we will have to investigate the inputs and create logic for a larger range of inputs.

Inputs to the level builder are the touch location and paths from the touch screen input processing. The only output is the game map.

Touch Screen Input Processing

We plan on using a touch screen as the input for the level builder. It will read in data from the touch screen and process it to give coordinates of touches to the level building module. This requires attaching an external component and learning its communication protocol.The difficulty with this will be obtaining and using a new component and we can mitigate this by getting the component early and researching possible library for its communication protocol exists that we can leverage.

The input to the touch screen input processing is the information from the touch screen. The output is the touch location and paths for the level builder.

Project Timeline

Week	**Rachael will be doing**	**Kim will be doing**
Oct 29th	Game play (module 1)	Graphics (module 2)
Nov 5th	Game play (module 1)	Powerups (module 3)
Nov 13th	Enemy AI (module 4)	Game Menu (module 5)
Nov 20th	Touch input (module 6)	Easter Egg (module 5)
Nov 27th	Slack/documentation	Level builder (module 7)
Dec 4th	Slack/debugging/ documentation	Slack/debugging/ documentation
Dec 11th	Showtime	Showtime