PRIMEDesigner15 Documentation and Research Report

Paul Curry, Dennis Orzikh

**Table of Contents:**

[What is PRIMEDesigner15?](#h.t8tu12o16pfi)

[Soluzion Client and Template System - Implementation](#h.19ndgbye55f0)

[Brython](#h.6br3oguwtnoc)

[“The State”](#h.2v2qmerptcz8)

[Operators](#h.8pgzyp2iw6qp)

[Functionality](#h.dn9k5woghphq)

[Implementation](#h.59tlpeohcpj3)

[JSON Exporting Operator](#h.fbvxxpmaieb0)

[Architect Role](#h.pnt7sw3mmylc)

[Display](#h.ocmw0j4yt6nu)

[Functionality](#h.mxf2d3huij1b)

[Implementation](#h.fs2846kb9fs9)

[Doors](#h.ctr2sbir019w)

[Functionality](#h.c5w24flswey1)

[Implementation](#h.1hjvj2fzrqca)

[Puzzles](#h.otmvsrtdgai8)

[Functionality](#h.41w9422i4tc)

[Implementation](#h.5rnhvm4ih90h)

[Ambient Audio](#h.uh03kno8y85z)

[Functionality](#h.xu9452bqsjzf)

[Implementation](#h.mwdysoesr5r)

[Wallpapers](#h.5u6nbb16spcs)

[Functionality](#h.wdmj0khgixxo)

[Implementation](#h.s1rmiujuou3n)

[Image Puzzle Designer](#h.sq6g2nrzkro1)

[Display](#h.qyycig13jvns)

[Functionality](#h.cpiyetpo5n01)

[Implementation](#h.fuo7bejjmtyz)

[Creating/Naming a Puzzle](#h.sbn05fswemzt)

[Functionality](#h.pgfla33fjj57)

[Transformations](#h.k9v000gktllq)

[Functionality](#h.esdv3wr70ess)

[Implementation](#h.5859z6cbigfs)

[Music Puzzle Designer](#h.sy49skpp8uqt)

[Display](#h.gn5tkd10ebj6)

[Functionality](#h.tbb7awnqcvtl)

[Implementation](#h.sg2mor93dxx2)

[Implementation](#h.s02cmju4gwai)

[Creating/Naming a Puzzle](#h.746wc8jxfeld)

[Functionality](#h.hepqbbvvztg2)

[Implementation](#h.c9x1l3n2yiya)

[Transformations](#h.bm4s2cec78hp)

[Functionality](#h.7ulozuce2l3)

[Implementation](#h.mx52t5e86vaa)

[Rule Designer](#h.dxw4ml1rckc)

[Display](#h.yieio3lyvv7)

[Functionality](#h.krqaz6c3zqbt)

[Implementation](#h.sbd807kac4vn)

[Creating and Editing a Rule](#h.78otkc6vil7e)

[Functionality](#h.tpvct675i82a)

[Implementation](#h.htuw4j7xq0l9)

[Challenges](#h.8al1vkp1aybv)

[Upgrading Brython](#h.tcuv6vk0ojf)

[Caman.js](#h.6l3l3zo583nr)

[Cryptic Errors](#h.qoqndcd0wki)

[Differences between JavaScript and Brython](#h.kfphvrmn0hc8)

[Introducing Asynchronous Operators](#h.1owtw21e00xw)

[Design Choices](#h.4x8v4ue4leda)

[Resources Used](#h.18661rw7u76g)

[Other Information](#h.xtojjdoia2j4)

#### 

# What is PRIMEDesigner15?

The Prime Designer research project primarily focused on the creation of a simple game designer featuring four distinct game designing roles that could be played in tandem. The game that the four designers work together to create would be a puzzle game with nine rooms connected in a square, and two types of puzzles, music and image puzzles. The four roles are Architect, Image Puzzle Designer, Music Puzzle Designer, and Rules Designer. The Architect places the doors which connect the nine rooms to together, the puzzles onto the walls, and specifies ambient noise and wallpapers for each room. The Image Puzzle Designer creates image puzzles out of jpegs or pngs of their choosing, and scrambles the images using pixel transformations that the end user has to unscramble during game play. The Music Puzzle Designer has a similar role to the Image Puzzle Designer, except they use mp3s to form their puzzles. The user has to transform the music back to its original form using inverse operations. Finally, the rules designer specifies abstract rules that determine when the user wins the game, which doors open or close, and when, and many other aspects of the logic executed in the game. In this write up we will focus on the functionality and implementation of each aspect of the game designer. This write up will serve as a record of the challenges we faced in the creation of our project, the choices we made, how our final client actually operates, and as documentation of the details of our code.

## 

## Soluzion Client and Template System - Implementation

The Prime Designer project consists of multiple parts: The Client and a main data structure management file that are both written assuming Brython is implemented, and a set of visualization and operator implementation files that can be written in any language. One would simply have to add a check for the language they are using into the main file, where it is currently checking for Brython. Thus, the current iteration of Prime Designer 15 uses Brython for the implementation files.

The Client is a new version of the Soluzion Client created by Steve Tanimoto, edited to allow for asynchronous operators (described further below). The main Prime Designer 15 files follow a template developed by Steve Tanimoto as well. Through these two standards, either part of the Prime Designer project is interchangeable with other projects following these standards. That is to say, any other operators-state-based brython project following the template can be displayed using the Soluzion client, and any other operators-state-based brython client can be used to meaningfully access the Prime Designer 15 project. However, at the time of writing, async support is unique to the client described in this write up, so transitioning to another client following these standards would still require some work in this regard.

# Brython

The second aspect of our research project was a study on Brython, a JavaScript interpreter for Python. Brython is a JavaScript library that directly interprets python files and runs them on the browser when linked within an HTML page. The syntax that makes up Prime Designer 15 is mostly Python 3, with JavaScript filling in gaps as necessary.

Brython has some advantages over JavaScript. Namely, its syntax is much cleaner than JavaScript as it doesn’t use brackets to define blocks and instead uses Python’s indentation based structure, this provides a much cleaner and easier to read code environment. Secondly, many of CPython’s built in modules are implemented in JavaScript via Brython. For example, the re module used for manipulation and analysis of strings through regex is supported. This allows someone who is more familiar with Python code than JavaScript to continue using the modules provided in CPython. This provides someone new to web development the ability to continue using tools they are familiar with as they explore new concepts. Finally, since so many modules from CPython are supported in Brython, a Python page that creates data structures in a Python compiler could also be used to create data structures in a browser, giving the language a degree of modularity.

There are some drawbacks however. According to their documentation, Brython is on average, about one third as slow as JavaScript. Some of the newer HTML5 features run so slowly that they cannot be written in Brython. For example, individual pixel manipulation on a canvas ran far too slowly on the latest version of Brython to be implemented. Thus, we wrote an external JavaScript solution that we imported in our Python code to handle the pixel manipulation necessary for the functionality of the Image Puzzle Designer. There are also some things you can do in JavaScript that you cannot do using Brython at all. For example, when trying to set a preserveAspectRatio attribute on an SVG pattern in the code for the visual display of the architect role, Brython would not recognize such an attribute. This forced us to write a simple three line JavaScript solution, patternPatch, saved in Prime Designer 15’s dependencies that defines an addAtttribute Function that uses the *setAttributeNS* function, which Brython does not recognize, but allows us to preserve the aspect ratio on an SVG pattern.

Lastly, Brython is still in active development, so support for Brython is currently not very extensive, especially compared to the widespread support of JavaScript. The Brython website itself provides some very detailed documentation, but falls short of being a full description of its capabilities. A lot of the examples in the documentation, like their SVG examples, stopped working after a particular update to Brython. The only community we could find that would provide in-depth support for Brython issues was a Brython google group, which was frequented by the developers of Brython, namely Pierre Quentel. While this group was certainly helpful, there was nowhere near the overwhelming support and solutions offered for JavaScript on W3schools, GitHub, and Stack Overflow. Yet this support for JavaScript was not built in a day, and, as computing speed increases, perhaps the performance issues of Brython will be overlooked, allowing Python to become the higher level language of web development, with JavaScript being a lower-level language.

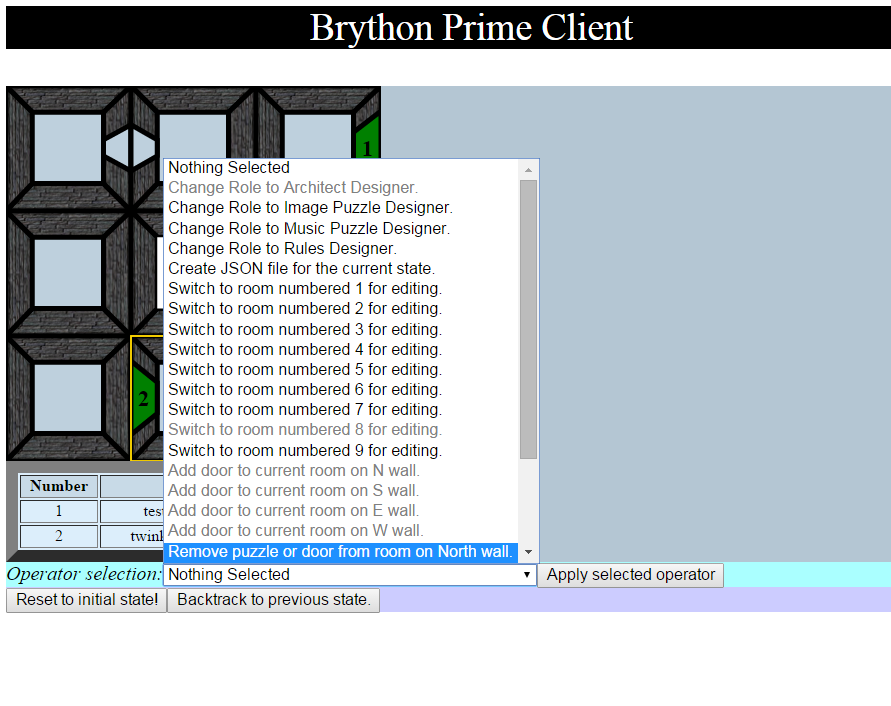
# “The State”

The state is a Brython dictionary object containing every data element in the game in its current state of development. This includes all of its rooms, wallpaper, puzzles, puzzle transformations, etc. There is an initial state that is set to contain 9 rooms with 4 walls each, and with the default brick wallpaper.

# Operators

## Functionality

A role can modify the state through its operators. Each time an operator is applied a new state is created. Thus each iteration of the game being designed is saved in a stack, and the user can click the “backtrack” button to discard the current state and move back to a previous operation. The user can also reset all their work to the initial state, that is, before any operators were applied. This method of storing all the iterations of the game design also allows for more advanced displays of the progress of development. For example a designer could be built that creates a tree-like display of states, and allows forks in the design process and individual selection of a state to load.

Each role has unique operators, but certain operators can be used by all four states. Every role can switch to a different role, export their state, and has a “nothing selected” operator that serves as an operator that does nothing. The “nothing selected” operator is the first operator presented in the drop down menu. This is so the user does not accidentally apply an operator after hitting the apply operator button twice. A user cannot apply an operator that would be impossible or redundant to apply.

## Implementation

There are two types of operators. The Operator class contains a descriptive string (name), a function to determine it if is applicable in the current state (precond), and a function that deep copies a state and modifies it while returning or providing access to a new state from the original modified one (state\_transf). The second operator class, AsyncOperator contains all of the attributes of the Operator class and is solely used to nominally specify that the operator’s state\_transf does not directly return a new state, but instead contains a callback function that could potentially return a new state. The client specifies the callback function. This allows asynchronous actions like AJAX requests and HTML user interfaces to occur. An asynchronous implementation is the only way to accomplish this as otherwise, since JavaScript is single-threaded, the program would have to pause or inefficiently loop to receive input and the browser would stop responding. In addition if an operator was not asynchronous the only way for a cancel button to be implemented on an operator would be for the state\_transf to return an unaltered copy of the old state, which would work as a way to cause no changes, but would add a redundant state copy onto the state stack of changes. Thus the asynchronous nature of the operator allows its state\_transf to simply not call its callback function if it wants no new state to be created.

Operators are applied via an onclick event handler attached to a button in the client. The operator is chosen from an HTML select drop down menu and its state\_transf is called to give the client the new modified state. If the operator is of the normal class Operator, the client simply assigns the new state to the return of the state\_transf, otherwise the client passes a callback function that is called by the state\_transf function and contains a new state as an argument, which it then processes. The precond function is run for each operator and the operator is greyed out in the select drop down menu if its precond function returns false. The definition of the operator class is stored in a separate file, templateRoot, as operators are included in all projects following the template that Prime Designer 15 uses, and should not be defined by a client or main files.

## JSON Exporting Operator

In order to fully encapsulate PrimeDesigner15, a user of the program should be able to save the game they are designing. Each of the four roles is able to apply this operator, create\_json\_file, which saves the current State in JSON format in the *PrimeDesigner15* folder under the *JSONOutputs* sub-folder. Although it does not need to “wait” for any further input or processing before running, It was made an AsyncOperator to make use of the optional state return functionality. Saving the JSON of the current state does not change the state in any way, so there is no need to create a new iteration of the state.

create\_json\_file’s state\_transf function is create\_json. It is within create\_json that the State is converted to a JSON object and then sent through an AJAX request to jsonPatch.php. Every class defined within the main data structures file (Room, Rule, etc) has an encode function which returns a list or dictionary containing all valuable information from the object it was called on. This process converts all important data from Python object attributes to elements within generic data structures that abide by JSON formatting. Through calling encode on all objects within the current state, create\_json creates a dictionary, stateJSON, that is very similar to the State but just formatted for exportation.

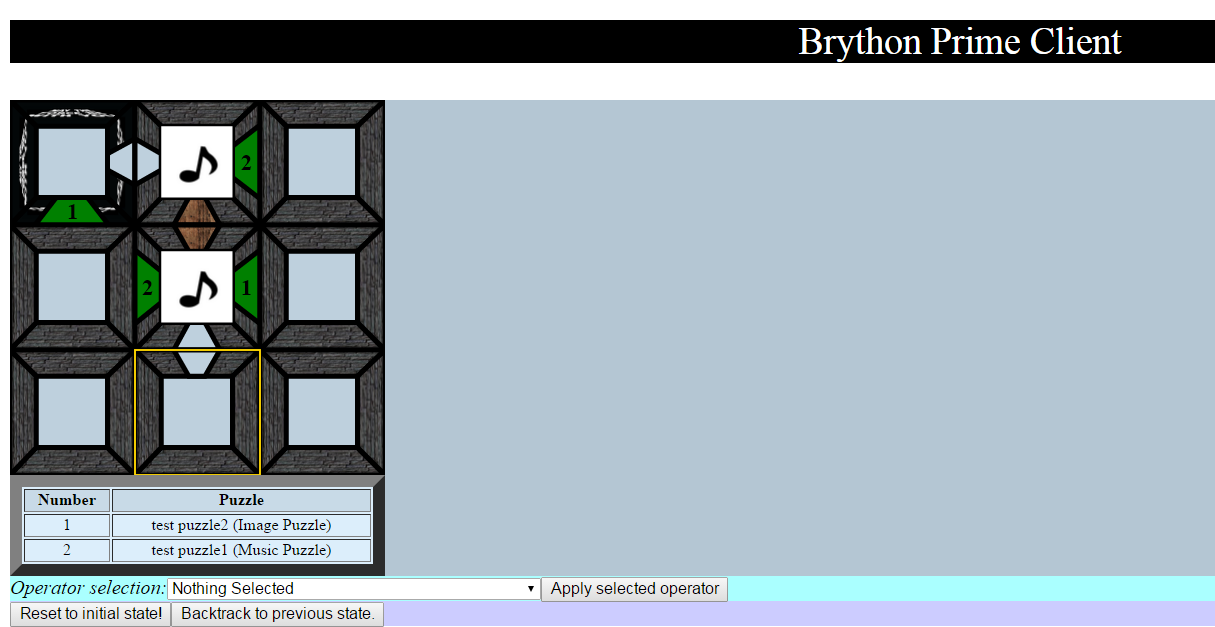
stateJSON also includes metadata from Prime Designer 15, such as the current Soluzion Client version, the Prime Designer 15 version, and the date on which the JSON was created. The current date is found through Python functionality that was carried over into Brython through the datetime library.

Once all information about the state is compiled as a JSON object, Brython makes an AJAX POST request to jsonPatch.php, as mentioned above, containing the JSON. This PHP script will always save the passed JSON as “stateJSON” in the JSONOutputs sub-folder. If a file with that name exists, it begins to add numbers to the end to avoid overriding previous files. This allows any number of JSON files to be saved without hassle.

# Architect Role

The architect role designs the physical structure of the game. What the player can see and interact with is defined by what the Architect chooses to place in the game. In this section we will define each object the Architect can place and explain our design and implementation choices related to that object.

## Display



### Functionality

The Architect is provided with a top down view of the nine rooms. The Architect picks which room to modify, and then can add various objects to that room. They can see, and change, the wallpaper on each of the four walls. If they add a door they can see it on the state display as a wooden door, open or closed depending on its initial creation. If the Architect adds ambient audio to a room, a musical note is placed on the floor of the room. The architect is also provided with a list of the puzzles available to be placed in the state and whether they are music or image puzzles. The list also provides each puzzle with an index number. When a puzzle is placed on the wall of a room a number is displayed in the center of a green rectangle on the wall informing the architect which puzzle they placed. The walls are differentiated using cardinals (North, East, South, West) for the Architect when they choose which wall to place an object onto.

### Implementation

The display for the architect role is an SVG object appended to the GUI. It is created in the visualization file. The walls are SVG polygons filled with a pattern that links to an image path on the server. There is coordinate information stored within the objects defined in the *PRIMEDesigner15* main file, but they are floating point numbers between 0 and 1. The coordinates are scaled up in the *PRIMEDesignerVisForBrython* visualization. In order to get the image to correctly fill the pattern and then the polygon, a technique found on StackOverflow was used that involved placing the image as a 1x1 pixel SVG image element inside the pattern. Adding the preserveAspectRatio attribute to the SVG image would allow the image to correctly fill the polygon. However, in order to have the most descriptive display possible, the wallpaper should be facing the vanishing point in the center of the room, creating a true top-down view. Thus, the program has to rotate the image in the pattern. This was confusing at first because adding a rotation by 1 pixel appeared to make the image disappear. This turned out to occur because the rotation was being applied to the image before it was scaled up to fit the polygon, thus a 1 px rotation would correspond to a huge rotation of the image element. After making the rotation have floating point values, and adding a translation to fit the image within the polygon, the wallpaper correctly fit in the wall polygon. The objects within the rooms are all pointers to instances of a particular class, so the visualizer renders them in an object-oriented fashion. Render\_state\_svg\_graphics draws all the rooms through a drawRoom function, which in turn calls a drawWall function. It continues down the chain of detail to the simplest pieces of data which are then rendered onto the SVG board. The only elements in the architect role not drawn this way is the puzzle list and the selected room outline.

## 

## Doors



### Functionality

The architect role can place doors that connect rooms in the initial state to each other. A door can only be placed on a wall if the wall is free of any objects (another door or a puzzle), the other side of the wall is also free of any objects, and if the door would open into a space within the game area. Doors can be specified as initially open or closed when created. The architect can also delete doors.

### Implementation

Unlike rooms and walls, there is no door object. Instead a door consists of two boolean attributes in each wall: whether it has a door (wall.hasDoor), and whether the door is open (wall.doorOpen). If the wall does not have a door, wall.doorOpen is null. Doors are drawn on each wall that has .hasDoor == True. To place a door, the architect must apply one of the create\_new\_door operators, as there are multiple created using a Python syntax list comprehension to cover all four walls. This operator is asynchronous and passes a callback function to the operator’s state\_transf. The state\_transf uses an imported function from PRIMEDesignerVisForBrython15 that uses a pop up HTML <div> to give it access to the user's decision on whether or not the door should be open, and passes it a callback to process the decision. This callback method of passing a function from PRIMEDesigner15 to a function within PRIMEDesignerVisForBrython15 to process gui outputs is common to all AsyncOperator operations involving GUI elements.

## Puzzles

### 

### Functionality

The architect role can place all puzzles created by both the image puzzle and music puzzle designers. A puzzle, unlike a door, can be placed on a wall separating two rooms. We made the choice to allow multiple copies of the same puzzle to be placed in the state by the architect because we wanted to allow the architect more freedom in designing the game. This allows, for example, making symmetrical states that could be used for a “race to the finish” type game. If a puzzle designer makes a change to the puzzle that was placed multiple times, each copy of that puzzle will be modified. The architect can delete puzzles from the game in development. Just because a puzzle is deleted from the game board, does not mean it is not still available in the list of available puzzles to be placed. Only the image/music designer can permanently delete image/music puzzles from the list of available puzzles.

### Implementation

While there are puzzle objects, a puzzle object is not directly referenced or placed within a wall. Since both music puzzles and image puzzles are a dictionary of unique puzzles names to actual puzzle objects, the wall only needs to store a wall.puzzle attribute that contains a string that is a key to a puzzle in one of the dictionaries. This allows us to save some space as well as avoid working with direct pointers of objects, i.e, a wall doesn’t need to store a whole puzzle object, just a string.

## Ambient Audio



### Functionality

Ambient audio allows the architect to enhance the setting of a room via audio that plays whenever the user enters a room. Ambient audio is controlled by the architect as opposed to the music puzzle designer because only the architect could match their ambient audio with the wallpaper they chose for each room. For example, it is common in video game levels that take place in a cave to have ambient audio of water dripping, wind howling ect. Ambient audio is placed room by room. To place ambient audio in a room, the architect selects the “place ambient audio in current room” operator. When this operator is applied the architect is prompted for the url path to a music file. If the music file is valid, the architect sees a musical note appear in the room where the ambient audio is located. To play the ambient audio, the architect can click on the musical note, turning it a shade of blue, When the ambient audio is finished playing, the musical note turns white again. The ambient audio finishes playing when the user clicks on the musical note again, or ten seconds pass.

### Implementation

To play the ambient audio the library wad.js is used. A Wad object is constructed within the playAmbientAudio function of PRIMEDesigner15MusicForBrython the file that handles audio. The Wad object allows a relative url to a music file as an argument in its constructor and contains play and pause functions to play and pause the playing of that music. playAmbientAudio places its generated Wad object into global storage and plays it. If the function is triggered again with a url that is the same as the one used in the global Wad, then it will play Wad otherwise it will generate a new Wad with the different url and play it. This allows the function to have a memory of one ambient audio element, so loading does not have to occur if the architect plays an ambient audio element.

Playing the ambient audio is done through an onclick attached to the musical note patterned svg rect with the event function ambientAudioTrigger. Since ambientAudioTrigger requires the room and svg element ambientDiv, a nested function has to be used to give the event function access to those parameters. In this case, ambientAudioTrigger returns a function, namely ambientAudioTrigger2, which actually processes the data with the parameters passed to ambientAudioTrigger. When the onclick is attached to ambientDiv, ambientAudioTrigger is called, returning a function that will fire after the user clicks on ambientDiv. This method of nesting a function *A* that returns a function *B* that utilizes parameters of *A* is used in other parts of the code.

When ambientAudioTrigger2 fires, the pattern on the ambientDiv is analyzed to see if the audio is still playing as there is no way to check a Wad object to see if it is currently playing audio. If the fill is of the pattern the designer sees when audio is playing, then the audio is stopped and ambientDiv’s pattern is set back to default. If the fill is of the audio not playing, then a timer is set using the Brython *timer* module based on the python module of the same name. The timer fires the stopPlaying function after ten seconds (declared as a constant), the pattern on ambientDiv is updated to signal playing, and the playAmbientAudio function of PRIMEDesigner15MusicForBrython is called.

## Wallpapers



### Functionality

Each wall has a default wallpaper set to a stone wall (wall.jpg). To replace the wallpaper of a room, the architect selects “Add wallpaper to room” from the operator drop down menu. They are then prompted for a url to an image. If the url is valid, the image is placed on all four of the walls in the selected room. The images on these walls are angled to face the vanishing point in the middle of the room.

### Implementation

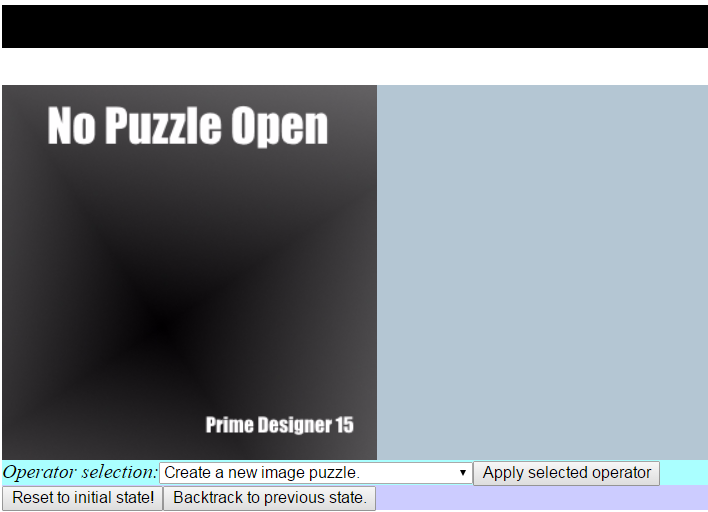
Each wall contains an attribute wallpaper that is a url string. This string is used in PRIMEDesigner15VisForBrython to define a pattern with its CSS style width and height set to “100%” that is appended to the gui and used in the fill of the wall svg polygon. Simply creating a pattern with a url specified is not enough to have the image from the url fit nicely into the polygon nor make it face the vanishing point in the center of the room. In order for this to occur, several tricks are employed. Firstly, the patternContentUnits of the pattern is set to objectBoundingBox using the function addAttribute defined in a 3 line JavaScript file patternPatch.js. This JavaScript defined function must be used because of an error with Brython not recognizing the specific CSS attribute. Next, an svg image is created with its x and y values set to 0 and its dimensions set to 1x1 units. A specific transformation is applied to the image depending on which side of the room it is on to make the image face the center of the room. The attribute preserveAspectRatio is set to “none” using addAttribute. This allows the image to cleanly fill the wall polygon whilst still facing the correct orientation.

The state\_transf of the “Add wallpaper to room” operator is add\_wallpaper\_to\_room. It uses a JavaScript prompt to get a url, checks to see if the url is valid with url\_is\_valid, and, if so, copies the state and assigns the wallpaper attribute of all four of the walls in the selected room to the valid url.

# Image Puzzle Designer

The image puzzle designer uses images to create puzzles based on image manipulation. The designer can create and delete image puzzles, which are added to the list of available puzzles to be placed on the game in development by the architect. When the image designer creates an image puzzle via uploading an image, they start with the image unaltered and add pixel transformations to create a new image. In the game player, which is in development, if the player encounters an image puzzle, they can attempt to reverse the pixel transformations to get back to the initial image and “solve” the puzzle.

## Display



### 

### Functionality

The first thing the image puzzle designer sees is a black box with text that reads “No Puzzle Open”. When the designer chooses to create a new image puzzle they see their chosen image appear where the black box was. The user can choose operators just like any of the other role, via a drop down menu. When an operator is applied, an image transformation is applied to the image. The designer sees the resulting manipulation of the operator on the image.

### Implementation

The image puzzle designer uses a Canvas HTML5 element. This is the best element for the job because the canvas element has built in support for displaying uploaded images and performing pixel manipulations on them. The canvas object itself is created within Brython in the visualization file’s render\_state function.

Due to computational limitations with Brython, the pixel manipulation required for this role has to be done in JavaScript. Otherwise, it would be too slow and impede the user. For this reason we created a JavaScript class called CanvasManager which, as its name says, is intended to manage the Canvas. An instance of the CanvasManager is created within Brython using the JSConstructor function, and this allows us to call methods of the CanvasManager instance using Brython. Even though we only use one instance of CanvasManager at a time, this object-oriented approach allows us to only have to import from JavaScript once, instead of for every functionality the CanvasManager holds. As well, it limits the number of parameters that need to be passed into CanvasManager, streamlining the process.

As mentioned above, originally the Canvas holds a dummy image informing the designer that no image puzzle is open. Once they create an image puzzle, the new CanvasManager instance is passed the URL of the new image, which then alters the Canvas to show the new image. Every time a new State is rendered, an new instance of CanvasManager is created.

## 

## Creating/Naming a Puzzle



### Functionality

When the designer chooses to create a new image puzzle, they are given a text prompt to choose a URL path to an image. If the url is valid the designer sees their chosen image appear where the “No Puzzle Selected” box was. The name of the puzzle is assigned based on the name of the image file imported. If the designer imports the same image file multiple times, a number is placed next to the name that represents which copy the puzzle is. If the designer chooses to rename a puzzle they are also given a text box prompt to enter in a new name for the puzzle.

**Implementation**

The operator to create a new image puzzle is an AsyncOperator so that the user can click cancel on JavaScript’s built in prompt box, referred to in the code as window.prompt. After the user enters the URL, it is validated using the Python open function which is included with Brython. Once validated, a new state is created with the image as a new puzzle. The name of the puzzle is defaulted to the name of the image selected, plus a number depending on if it is the first puzzle of that name or not.

Renaming an image puzzle is simple. The user first selects the rename puzzle operator, whose name also includes the name of the puzzle that will be renamed. The state\_transf uses window.prompt to prompt the user for a new puzzle name. Due to the dictionary nature of the two puzzle lists, all puzzles, music and image, must have unique names. The check\_puzzle\_names function confirms that the new name the user entered is unique to all puzzles. If it is not, the function asks the user to enter in a new name. If the user enters a unique name the function then uses the dict.pop(key) Python function that removes the specified key from the dictionary if it is within the dictionary, and associates the new name with the selected puzzle as a key value pair in the image puzzle dictionary.

## 

## Transformations

### 

### 

### Functionality

There are five transformations and three inverse transformations the image puzzle designer can use to “scramble” the images in their image puzzles. Each one is applied like any other operator in the game designer. The image puzzle designer can horizontally or vertically flip an image, shuffle its pixel columns or rows, or perform a pixel crossover. The inverse transformations for shuffling the rows or columns, or for pixel crossover, each undo one application of their respective transformations. Each transformation can be applied multiple times and they can all be layered on top of eachother.

### Implementation

When the designer adds a new transformation to their image puzzle, it is added to a list of strings, the transformList, which is saved by an instance of the ImagePuzzle class. When the current state of the image puzzle is to be rendered, the function drawImagePuzzle waits for the CanvasManager instance to finish preparing the Canvas, and then, using a call back, reads through the puzzle’s list of transformations and chooses the appropriate transformation method from the CanvasManager instance to run for every transformation listed.

The process within the CanvasManager methods requires the JavaScript getImageData method of the Canvas. This method returns a one dimensional array that represents the three dimensions of the image: the width, the height, and the color channels (RGBA). Two copies are made of the returned list. One is altered to contain new data, and the other captures this data for returning to the Canvas using putImageData. Since every transformation edits not only every single pixel in the image, but also every pixel attribute, every transformation requires a triple nested for loop that goes through every column along the width of the image (the X axis), every row along the height (the Y axis), and the RGBA of every pixel (the Z axis). Taking the horizontal flip method as an example, the final formula inside the triple nested for loop would look like this:

toBeSent[(row \* width + column) \* 4 + RGBA] =

original[(row \* width + (width - 1 - column)) \* 4 + RGBA]

Or in cartesian terms:

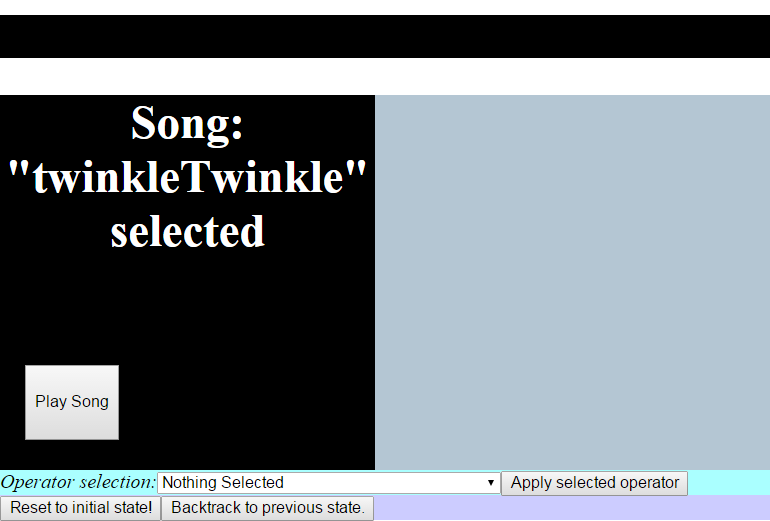
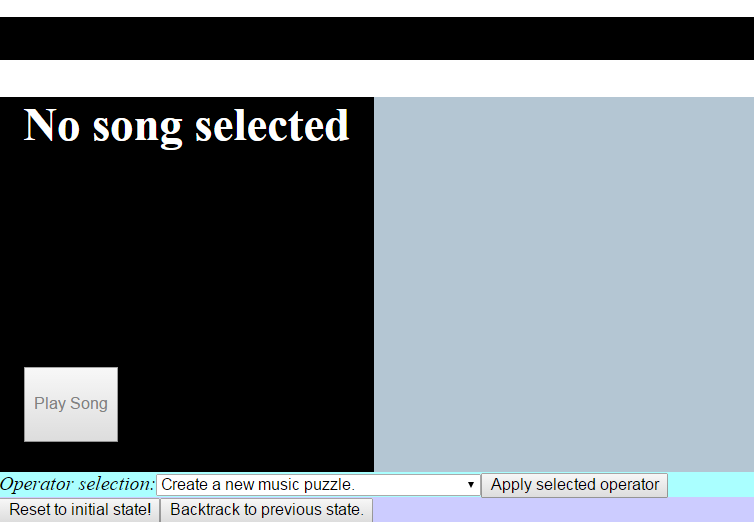
toBeSent @ (X,Y,Z) = original @ (width - 1 - X, Y, Z)

Once the CanvasManager method finishes going through every pixel, it places the completed, new image data onto the Canvas attached to the CanvasManager instance using putImageData.

# Music Puzzle Designer

Like the image puzzle designer, the music puzzle designer creates puzzles that the architect can place within the game in development. Instead of pixel manipulations as transformations, the music puzzle designer rearranges individual notes and applies audio filters to their sound.

## Display



### Functionality

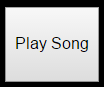
The music puzzle designer first sees a black box with the text “No song selected” and a “Play Song” button that is greyed out. Once a music puzzle is created, the “Play Song” button becomes active and the user can click it to hear a series of notes. The text at the top of the black box also reads “Song: [Song name] selected”. When the “Play Song” button is clicked it becomes greyed out until the music finishes, then it becomes enabled again. The music puzzle designer can switch to a different music puzzle that they have created by applying different operators that are labeled “Selected puzzle number [chronological index number of the music puzzle], [Name of puzzle] for editing.”

### Implementation

The UI for the music puzzle designer is constructed with simple HTML5 elements. Starting from the top down, a div with the ID “musicDisplay” is appended to the GUI. Within this div is a button with the ID “playbutton” and an HTML5 p.The p is changed in the intitial\_render function of the client BrythonSOLUZIONClient each time a new state is loaded.

Music is played using a series of Wad objects created using the library wad.js. Each time a new state is loaded the “Play Song” button is unbound of all event handlers then bound with an onclick, handlePlayButtonClick imported from the file that handles all audio in Prime Designer 15: PRIMEDesigner15MusicForBrython.py.

**Playing Music**



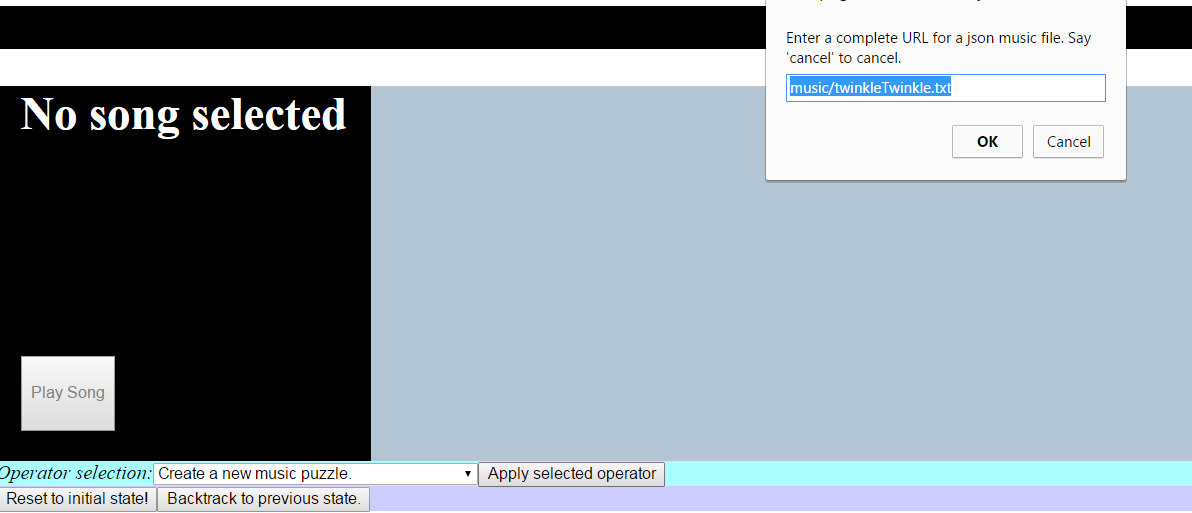
### Implementation

The playSong function in PRIMEDesigner15MusicForBrython handles the playing of notes contained in a MusicPuzzle object. The function takes a state as a parameter, and uses the “Selected\_Music” of that state as the music puzzle which it is to play. After transformations are applied, a wait variable representing the amount of time before a note begins and a timeLimit variable, specifying the maximum time a song can play for, are set.

Each note is read out and the default Wad object paino, instantiated at initialization, is used to play each note. A Wad object can take parameters in the form of a dictionary in its play function. The wait, pitch, and hold values taken from the note in notes are passed into the play function called from paino. The wait period is incremented based on the “wait” value of the note that is being played. In such an implementation, the note’s wait time is relative to the note before it. For example, in the song displayed in the JSON below, in the implementation of creating a puzzle, the first note would play after 0.5 seconds. Then the next note would be played using the wad.js object specified with a wait of 0.5 + 0.5 = 1 seconds. Thus 0.5 seconds after the first note is heard, the second note will be heard.

We chose this delimitation of wait times because it seemed to make the most sense, as a user creating their own sheet music could add notes in the middle of their sheet music without having to update the entirety of the file. The delimitation is also the closest to how we read sheet music, instead of reading a note based on its time from the beginning of a song, a person usually references the wait time between the previous note. To create a chord (have multiple notes play at the same time), a user can specify the wait period of a note to be 0, which means the note would wait 0 seconds after the note before it to play, effectively playing both notes at the same time. If the waits increment to a value more than timeLimit the music stops and a JavaScript alert informs the user that their song is too long to play.

## Creating/Naming a Puzzle



### 

### Functionality

To create a music puzzle, the music puzzle designer applies the “Create a new music puzzle” option from the drop down list. A popup appears prompting the user to enter the URL for a sheet music file. If the URL is valid, then the puzzle is created and added to the list of puzzles the music puzzle designer can edit with the name of the file its URL pointed to. If the same file has been imported twice, the puzzle has a number attached to its name that states which copy it is, like in the image designer. If the designer wishes to rename a puzzle, they can select the “Rename selected puzzle” operator from the drop down menu. The designer is then prompted for a name. If a music or image puzzle already exists with the name the designer entered, the designer will be prompted again for a name and informed that a puzzle already has the first name they chose.

### Implementation

The music puzzle class contains a list, notes, which will hold dictionaries created from loading JSON. Each dictionary contains information for one note. Another list, transformList, is filled with strings delimiting the transformations used on the puzzle.

The “Create a new image puzzle” operator is asynchronous. Its state\_transf is the function create\_music\_puzzle, which first prompts the user with the built in JavaScript prompt function for a complete URL to a JSON music file. If url is validated via url\_is\_valid, an AJAX request to read the JSON file is created and sent with the function requestSuccess called with a name derived from the url of the JSON file to bind requestSuccess2 to the completion of the request. There is also a call to show\_loading, a function defined in the visualization that displays a popup div containing a loading gif so that the designer knows that PrimeDesigner15 is working.

requestSuccess returns requestSuccess2. This method of function nesting, which allows event based functions to utilize passed parameters, is discussed earlier in the ambient audio implementation section. If the AJAX request is a success its status is either 200 or 0, otherwise an error message is printed if the AJAX request is not successful. The JSON files that the function reads contains information to play a song. The JSON is a dictionary with the key “notes” that contains several other non explicitly named objects. Each note contains a “wait” mapping to a floating point value, a “pitch” which is delimited by the note letter and in integer representing the octave the note is in, and a “hold” key mapping to a floating point value. An example of the JSON is below:

{"notes" :[

{"wait": 0.5, "pitch" : "E3", "hold" : 0.5},

{"wait": 0, "pitch" : "A3", "hold" : 0.5},

{"wait": 0.5, "pitch" : "A3", "hold" : 0.5},

{"wait": 0.5, "pitch" : "A3", "hold" : 0.5},

{"wait": 0.5, "pitch" : "A3", "hold" : 0.5},

{"wait": 0.5, "pitch" : "G3", "hold" : 0.5}

]}

After requestSuccess2 fires, the state is copied. The JSON is parsed with the json.loads function of the json module and the list of notes is passed to the constructor of a MusicPuzzle. A name for the puzzle is given by the upper-level requestSuccess function. The name is then mapped to the new puzzle object and placed within the “Music\_Puzzles” dictionary of the new State. The loading gif is hidden with a call to hide\_loading and the callback function given to the state\_transf is called to send back to the new state.

## Transformations

### Functionality

A transformation is added to a selected music puzzle via separate operators with descriptions of their transformation. The transformations can apply filters to the notes being played, rearrange the notes, and change the tempo of each note. When a transformation is applied to a puzzle, that transformation is permanent and stacks with any other transformation applied to the puzzle.

### Implementation

When a transformation operator is applied, a new state is copied and a descriptive string is placed in the transformList of the “Selected\_Music” puzzle of the State. When playSong fires after a user clicks on the “Play Song” button, the transformations are applied. First the default tempo and pitchChange are set. Depending on the transformation, these values may be altered.

“increasePitch” and “decreasePitch” are both examples of the strings stored in transformList. They increase and decrease the pitchChange value respectively. When the default Wad object, piano is played, the “pitch” string is converted into a frequency number via the large Pitches dictionary that is built in to wad.js. pitchChange is then added to the frequency value which is mapped by the “pitch” key in the dictionary passed to piano.play. “increaseTempo” and “decreaseTempo” are transformations that change the tempo variable. A note’s wait period in during the playing while loop are multiplied by this value.

The “shuffleNotes” and “reverseNotes” operators require special processing of the notes list before they can be applied. Since there is no JSON delimited chord there is nothing preventing these transformations from breaking up chords which would make the song impossible to solve in “shuffleNotes” and incorrectly play in “reverseNotes”. In order to fix this, the notes are grouped into separate lists that represent chords via the groupIntoChords function. For these functions a separate list of lists, notes2, is created, and the transformations modify the chords stored with notes2. When these transformations are done being applied, the notes of notes2 replace those of notes via the readChords function.

# Rule Designer

The rule designer creates and edits rules for gameplay. Rules affect how the player of the completed game board interacts with their environment, their progression, and eventually how they beat the game.

Every rule may consist of any number of conditions, and any number of actions. Conditions are events that need to take place in order to trigger a rule. Once a rule is triggered, all attached actions should occur. However, the rule designer is not responsible for objects associated with their rules. For example, the designer can create a rule with an action that involves opening a door on the south wall of room 5, but this door may or may not exist at the time of the rule’s creation. This freedom does not limit the construction of the game to a chronological order, as otherwise the rules designer would have to wait till the architect is finished placing objects to begin contributing to development.

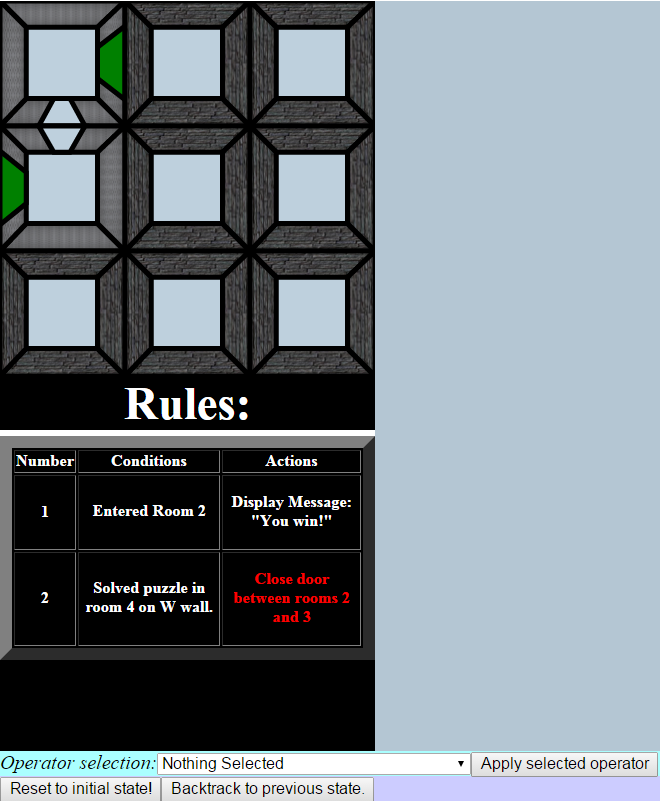
Conditions available:

* The player entered a room
* The player had a certain number of points
* A certain amount of time has elapsed
* The player solved a puzzle in some particular place

Actions available:

* A particular door opens or closes
* A sound is played
* A message is displayed
* A puzzle is unsolved
* The player gains or loses a certain number of points
* The game ends

## Display



### 

### Functionality

The rule designer’s display consists of two parts, the rooms display and the rules display. The rooms display is the same one that the architect sees. Rules are displayed in a grid that splits them up into lists of their Conditions and Actions. If there are no rules yet, the message “There are no Rules” is displayed in place of a grid. The grid assigns each rule a number, and this number is used to identify each rule’s editing operator. As mentioned above, a rule may include an action or condition that is not able to be attached to anything in the current game state; that is, an action or condition may be inapplicable. The display grid individually colors every action or condition to display their applicability. White conditions or actions are applicable, red ones are not.

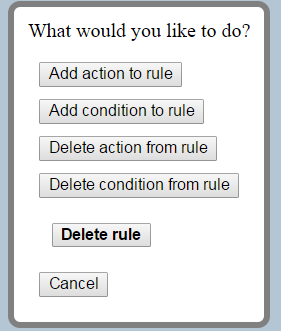
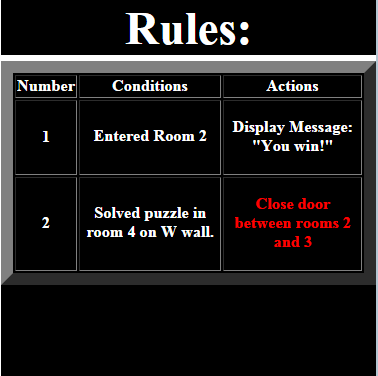
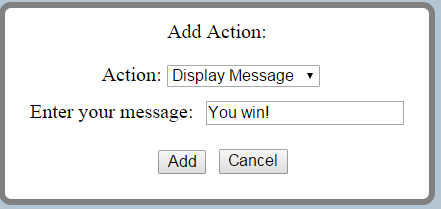
### Implementation

The rules display calls the same drawRoom function as the architect display. The only difference between this part of the display for the rule role and the display of the architect role is that the rule designer has no operators available that will alter what is displayed for the rooms.

The rule display grid is actually an HTML table that is created and populated in the populateRuleDisplay function. This function builds up an HTML <table>, giving each rule a number and listing its conditions and actions in separate columns. The table is placed within a <div> with the id of “tableWrapper” whose “position” attribute is set to “relative” and “height” set to “auto” to allow the list to fill infinitely and scroll without moving off the page. Choosing the color of each rule element (conditions and actions) is based on the app attribute of the associated ruleElement object attached to the Rule. Every ruleElement’s applicability is determined when it is first created, and updated every time a change is made to the game state that could affect applicability (door added or removed, puzzle added or removed, etc). This check is done in the check\_rules function.

## 

## Creating and Editing a Rule



### 

### Functionality

The rule designer first creates blank Rules using the create\_rule operator, and then add actions or conditions to the rule using the edit\_rule operator. edit\_rule also allows the designer to delete an entire rule or parts of one from the state.

When the rule designer applies an edit\_rule operator, a menu pops up asking them what they want to do. They may add an action or condition to the chosen rule, delete an action or condition, or delete an entire rule all together. Every choice is a button which the designer would click to proceed to a new menu, or they may cancel to exit out of the operator and make no changes to the state.The cancel option is available in every menu that may follow.

### Implementation

edit\_rule is yet another AsyncOperator, which allows cancelling functionality, and making no changes to the state. When the edit\_rule operator is applied, its state\_transf function, editRule, calls edit\_rule\_menu, an imported function from the vis file that creates the menu with all of the edit options. When a button that isn’t ‘Cancel’ is clicked, edit\_rule\_menu returns the designer’s selected edit through a callback, processEdit, within editRule. From here, the necessary function is called to show the user the next menu for their selection.

If they choose to add an action or condition, one of two very similar functions is called: addAction or addCondition. These functions call their respective imported vis functions, add\_action\_menu or add\_condition\_menu, and have a callback to process further menu results. Both of these menu functions then call create\_menu and a form creation function, add\_action\_form or add\_condition\_form, that creates a drop down for all possible types of action or condition, based on the two lists shown in the above Role section. When the designer chooses one of the drop down options, the menu is altered to show a new, secondary, drop down list (through aFollowUp or cFollowUp) that allows the designer to specify details for the rule, if necessary. For example, the “Player entered a room” condition would require the follow up question, “which room?”. Once PrimeDesigner15 has sufficient information, the rule element is added to the rule through a callback in the main data structures file (processAction or processCondition). In this same callback, copy\_state is called to create a new state, check\_rules is run to determine the new rule element’s applicability, and the new state is returned to the client through sendBack.

Although removing a rule element is functionally different from adding one, the only major difference lies in the processing function, processDelete, which pops a chosen ruleElement instead of appending one. Otherwise, the menu and callback structure is the same. On the other hand, deleting an entire rule requires much less specificity, so a processing callback is not even necessary. Once the option is selected from the editing menu, the chosen Rule is popped from the Rules list immediately.

# Challenges

## Upgrading Brython

During the development of the image puzzle designer we were running into speed issues regarding the manipulations of pixels on a canvas. The most frequent tip we got when asking on the Brython Google Group was to upgrade versions, as we were using Brython 2.2.1 instead of the most recent one which was Brython 3.1.3. After deciding to upgrade, we noticed that the scope and import systems had changed. In Brython 2.2.1 variables declared outside of a function are put into the global namespace and could be accessed by any .py file linked to the same HTML document. In Brython 3.1.3 the scope system had changed to be file specific, meaning that a variable declared within the broadest scope of PRIMEDesigner15 could not be utilized by PRIMEDesigner15VisForBrython unless it was explicitly imported, even if the keyword global was used in its declaration. This meant we had to write a lot of imports into our code which bound our files into chronological order of initialization. If any of the files are moved around in their initialization in the HTML page, the program will break, as certain files will not have access to globals they reference on startup. Strange unexplained things also occur with Brython’s imports, for example, placing an alert in a function that is imported from another file and ran once will output the alert twice as if it was fired twice. This might just be an exception with alert though, as all our imported functions do not fire twice unintentionally.

## Caman.js

Once we switched Brython versions, there was still the original issue of Canvas pixel manipulation. Although there was a noticeable increase in speed of our Brythonic pixel manipulation function, it was still far from acceptable for an end-user. It would take around five minutes to finish a manipulation on a 200 by 200 pixel image. At this point, image manipulation became the first functionality we deemed JavaScript was necessary to complete. We hoped to find an open-source JavaScript library that would be easy to use and readable, as PrimeDesigner15 was intended to be a project based on Python experience and not JavaScript. Based on this reasoning, we tried to implement a library called Caman.js. However, what seemed simple on the surface ended up being extremely difficult to interface with Brython. There was less support available for this library than even Brython, and the library itself had not been updated since 2013. With little understanding of how to even use Caman in a pure JavaScript environment, little progress was made connecting the library to our Brython functions. Caman would have also required asynchronous behavior, which we had not implemented at the time either. Eventually, we found that the more direct method of using JavaScript’s Canvas attributes and writing all the transformation logic ourselves would be easier. It also ended up running almost instantly as expected, unlike Brython with its five minute runtime.

## Cryptic Errors

Bug fixing with Brython was difficult mainly due to the fact that the error logs delivered to the console were often unintelligible or unhelpful. There was a slight method to the madness of the error messages. Often a syntax error within our code was greeted with an “uncaught error” alert from the console. The second and most confusing error message was this:

Error

at Error (native)

at Object.\_b\_.NameError (eval at $make\_exc (http://localhost/primeDesigner15/dependencies/brython.js:5785:11), <anonymous>:91:381)

at Object.$B.$NameError (http://localhost/primeDesigner15/dependencies/brython.js:5801:11)

at Object.$B.$search (http://localhost/primeDesigner15/dependencies/brython.js:4547:36)

at Object.eval [as replaceCurrentState] (eval at brython (http://localhost/primeDesigner15/dependencies/brython.js:3931:12), <anonymous>:295:13)

at Object.eval [as recieveNewState] (eval at brython (http://localhost/primeDesigner15/dependencies/brython.js:3931:12), <anonymous>:259:55)

at eval (eval at brython (http://localhost/primeDesigner15/dependencies/brython.js:3931:12), <anonymous>:335:51)

at HTMLButtonElement.<anonymous> (http://localhost/primeDesigner15/dependencies/brython.js:9253:58)

This error message would be returned for random syntax/name error mistakes. After rummaging around in the code for brython.js it appeared to be a javascript error being reported for the execution of an error handling function within brython.js itself. Despite the frequent occurrence of the above mentioned error messages, most NameError, IndexOutOfBounds, and import errors handled well by Brython and reported in the console. However, often the use of the console.log function and alert were required to bug fix our code. Stopping on exception would only take the console deep into brython.js.

## 

## Differences between JavaScript and Brython

The poor handling of errors led to a lot of trial and error situations when trying to understand differences between Brython and JavaScript. Although there is abundant and clear documentation for Python and JavaScript syntax, and it is easy to understand differences in syntax between the two languages, there is little support for things Brython does that Python would never do, such as handling of the HTML DOM. For example, in many cases changes to the style of an HTML or SVG element in Brython would work like so: myElement.style.display = “none”. In other cases, this would not make any changes at all, and one would have to say: myElement.elt.style.display = “none”. Either way could make sense conceptually, but the problem was that it was not consistent the way JavaScript is. Another example would be binding click handlers. Sometimes the .onclick property would be sufficient to add an event handler to the element, but other times it would not work without the more explicit .bind('click', myFunction). The freedom of choosing between different syntaxes for doing the same thing exists in JavaScript as well, but both methods would always work. With Brython, it was always a matter of analyzing unexpected behavior.

## Introducing Asynchronous Operators

PrimeDesigner15 is, as mentioned above, a project based on a template system invented by Professor Steve Tanimoto. The SOLUZION client, Operators, and many other aspects are based on this existing design. In all other projects using this system, it was perfectly reasonable for all possible actions to be individual Operators that always returned a new state. However, in PrimeDesigner15, the system ran into a complexity that it was not designed to handle: Actions which do not alter the state. For this reason we had to make an adjustment at a foundational level to the template, altering the SOLUZION client and adding new Operator functionality. At the same time, we wanted the new client to be backwards compatible with other projects following the template system. Our final solution was to implement the previously discussed AyncOperator, which inherits from Operator. The SOLUZION client will test the type of each Operator that is applied. If it is an AsyncOperator, it executes our new callback logic that does not require an immediate return from the state\_transf function associated with the operator. Otherwise, it will apply the Operator the same way old versions of SOLUZION did, achieving backwards compatibility.

## Design Choices

Since this was game designer the overall design aspects of Prime Designer 15 were focused on communicating information easily and providing easy user access to the functions of the game designer over aesthetics. The artistic design is meant to be non-intrusive, black and light blue is used to attract the user's attention to the game development screen. For doors, ambient music, and puzzles, simple image choices were made to communicate what that object is. Puzzles are delimited by their position in the list to the architect instead of via unique images on the game board, as it could be extremely difficult to determine which puzzle is which if two puzzles use a similar original file for scrambling. We tried to keep operators at a minimum, and often concatenated multiple operators into one operator, like with the “Edit Rule” operator in the rule designer which has the function of what was once 5 operators per rule. All of the buttons in the pop up menus have margins of about 10 pixels, to make the menus seem less crowded.

# Resources Used

* Brython version 3.1.3
  + <http://www.brython.info/>
* Vanilla JavaScript
  + https://www.javascript.com/
* Wad.js
  + https://github.com/rserota/wad
* PHP
  + https://secure.php.net/
* GitHub
  + <https://github.com/>
* Wamp
  + http://www.wampserver.com/en/
* Notepad++
  + https://notepad-plus-plus.org/

# Other Information

* Total commits to PRIMEDesigner15: 315
* Time in development: Apr 12, 2015 – Sep 22, 2015
* Total lines of our code: 3,590
* Code makeup: Python: 89.2%, JavaScript 10.7%, HTML 0.1%.