CS108, Stanford
Fall, 2008-09

Handout #36
Osvaldo Jiménez

Bunny World

Thanks to Nick Parlante for much of this handout

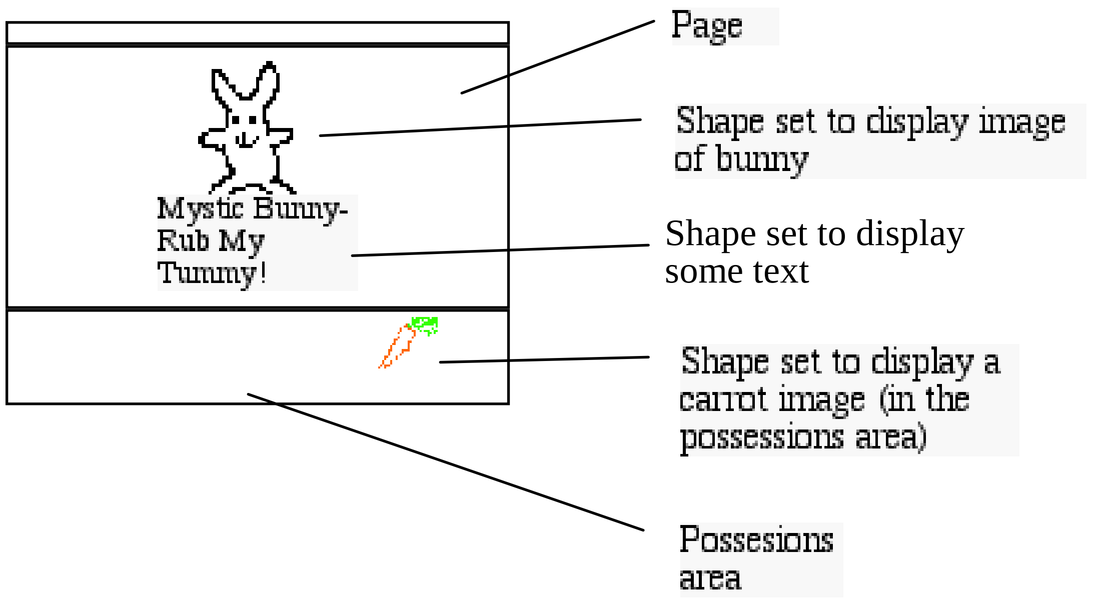
The Final Project gives you a chance to use the latest technology to solve a significant problem with a real deadline. When your Bunny World comes together, it's a moment of well earned joy. The whole project is due by **11:59pm on Wed Dec 3rd**, and then you can study for your other finals.

The goal for the final project is to implement an editor for a simple graphical adventure game. The game is called "Bunny World" after its most famous puzzle, but in reality, there's not much bunny in it. The project has two aspects: playing the game, which is relatively easy, and the game editor, which is a complete, graphical, document oriented, OOP/GUI project. This handout gives the ground rules for Bunny World— it defines the components and rules for the Bunny World format and lists the problems an editor needs to solve to be functional.

The Bunny World Ideas section gives suggested Human Interface (HI) and implementation ideas. Those will just be suggestions— any editor that enables its users to work with the Bunny World system as described in this handout is fine.

Part 1 — Bunny World Rules

This section introduces the components and rules that define Bunny World. Bunny World is not actually specific to bunnies in any way, it's really a generic graphical world populated with pictures and sounds. It's just preferable to use bunnies since they're cute and easy to draw. At a basic level, Bunny World operates like a low-budget version of Myst.


Mystic Bunny-
Rub My
Tummy!
Page
Shape set to display image of bunny
Shape set to display some text
Shape set to display a carrot image (in the possessions area)
Possesions area

The basic elements of the game are...

1) Pages

A page is a rectangular area with a white background that can contain a number of "shape" objects. The page area should be at least 500 x 300, and may be larger. Shapes or parts of shapes that fall outside the current page size are simply not drawn. The overall document is made up of many pages. Only one page and its contents are visible at a time. Certain actions within the program will which page is currently displayed. There is a special "page1" page where the game starts out when first run. Every element in the game, including pages, has a unique "name" string that identifies it. All the operations below that involve names or other text elements should not be case-sensitive.

2) Shapes

Each shape has several attributes:

Each shape has a name. The default names are "shape1", "shape2", ... but the user can change them to something more sensible.

Each shape belongs to a particular page, or the possessions area.

Like Whiteboard, each shape has a bounding rectangle that can be moved and resized. In the simplest case, a shape draws itself simply as a light gray rectangle.

In Whiteboard, there were separate shape classes, fixed at the time the shape is created. In contrast, in Bunnyword there is one shape class, and the rect vs. image vs. text distinctions (below) are made on the fly by changes in the state (ivars) of the shape. This is also a fine approach -- it's less structured and allows more dynamic behavior.

Each shape stores the name of the image that it can draw. If the image name is not the empty string, the shape draws the given image, scaled to fit the shape rectangle. If a shape does not have an image or the image cannot be loaded, then the shape should draw a light gray rectangle instead. Note that multiple shapes can refer to the same image by name. The shape does not "own" the picture, it just refers to it by name at the moment of drawing.

Each shape also stores some text that it can draw. If the text is not the empty string, the shape draws the given text, picking font size such that the height of font, in pixels fits within the height of the bounds rectangle. The text should draw as a single line of text. There should be a way to edit the font, bold on/off and italic on/off of the text. Use the font "Dialog" as the default (all Java runtimes at least have that font available.) If a shape has both an image and text, the text takes precedence -- the text draws and the image does not.

Each shape may be "hidden" in which case in "play mode" it does not draw and is not clickable. In edit mode, all shapes are shown.

Each shape may be "movable" in which case the user can drag it around in play mode.

Each shape has a block of "shape script" text, described below.

3) Possessions Area

Just below the current the page there is a visually separate "possessions" area. As the player or author moves from page to page, the possessions area stays constant, and the shapes in it just sit there. During play, the possessions area enables the player to carry objects from one page to another.

Visually, shapes should not rest touching the boundary between the current page and the possessions area. Shapes moved onto the boundary should move themselves to whichever unambiguous position requires the least movement. The shape should either move up into the page so its bottom edge is just above the boundary, or down into the possessions area so its top edge is just below the boundary. It's ok if the part of a shape is visually clipped off when it's in transit between the current page and the possessions area. It's ok if the shapes in the possessions area are simply placed where the user drops them and stay the same size -- typically the bottom part of the shape will clip since it's too tall for the possessions area.

4) Resources

The game document will use image and sound resources. The user adds the resources to the document by selecting them using a file chooser on the file system, so a resource will be identified by the path the user choose, such as `/usr/class/cs108/hw/bunny/carrot.gif`. Before the user has saved their bunny document, the program should load the resource from its original path as above. See the File Saving section below for how to handle resource as the file is saved.

The shapes refer to the resources by name for drawing and playing purposes. Each shape stores the name of one image which it may use to draw itself. Shape script code (below) can refer to any number of sounds by name. A newly created Bunny World document should be empty— it should contain one page, no shapes, and no resources.

5) Shape Script

Every shape has a block of "script" text which programs how the shape behaves during the game. The script is structured as a set of "clauses" where each clause is a sequence of words separated from each other by whitespace, and the whole clause is ended by a semicolon (;). The order of the clauses in the script is not significant. The words do not contain whitespace characters or semicolons. Shape script code and the names are not case-sensitive. The script code is interpreted at game run-time to control what happens. For example the clause..."`on click play boing.au goto home-page;`" within a shape means that when that shape gets clicked, it plays the sound named "`boing.au`" and then the game switches to display the page named "`home-page`".

Script Actions

There are five script primitives that perform actions in the game. Multiple actions may be combined in a sequence, in which case they execute from left to right. Generally the verb part of the action comes first, possibly followed by its modifiers.

`goto <page-name>`	Switch to show the page of the given name.
`play <sound-name>`	Play the sound of the given name.
`hide <shape-name>`	Make the given shape invisible and un-clickable. The shape may or may not be on the currently displayed page.
`show <shape-name>`	Make the given shape visible and active. The shape may or may not be on the currently displayed page.
`beep`	Play the system beep. Handy for debugging.

Script Triggers

Trigger clauses in a shape's script define how it behaves in response to certain events during the game:

`on click <actions>`	Defines actions when the shape is clicked. The shape must not be hidden and must not be in the possessions area. Executes the actions from left to right. If there are multiple on-click clauses, the first one found will execute and the others are ignored.
`on enter <actions>`	If the page this shape is currently in has just been "switched to" in the game, then perform the given actions. This trigger is really conceptually a page level option, but it's easier to implement it in a shape in the page. This can be used, for example, to play a sound whenever the user switches to a certain page. If the game was already at the given page, then the "on enter" should not trigger. If a page contains

multiple shapes with "on enter" triggers, they should all execute, but their order of execution is arbitrary.

`on drop <shape-name> <actions>` Defines actions when the shape of the given name is dropped onto this shape. For example, if a bunny shape contains the clause "on drop carrot hide carrot play munching;" -- then when the shape named "carrot" is dropped on the bunny, then the bunny will make the carrot disappear and play the munching sound. A shape may contain multiple "`on drop`" clauses to respond to having different shapes dropped on it. During drag-n-drop in play mode, the destination shape gives visual feedback to indicate if it has a matching on-drop (described below).

Play Mode

During the play of the game, pretty much all the action is driven by clicking on objects which executes their "`on click`" triggers which play sounds, switch pages, etc. The only other action the user can initiate is dragging movable objects around.

Switching to play mode, starts by showing the page named "`page1`".

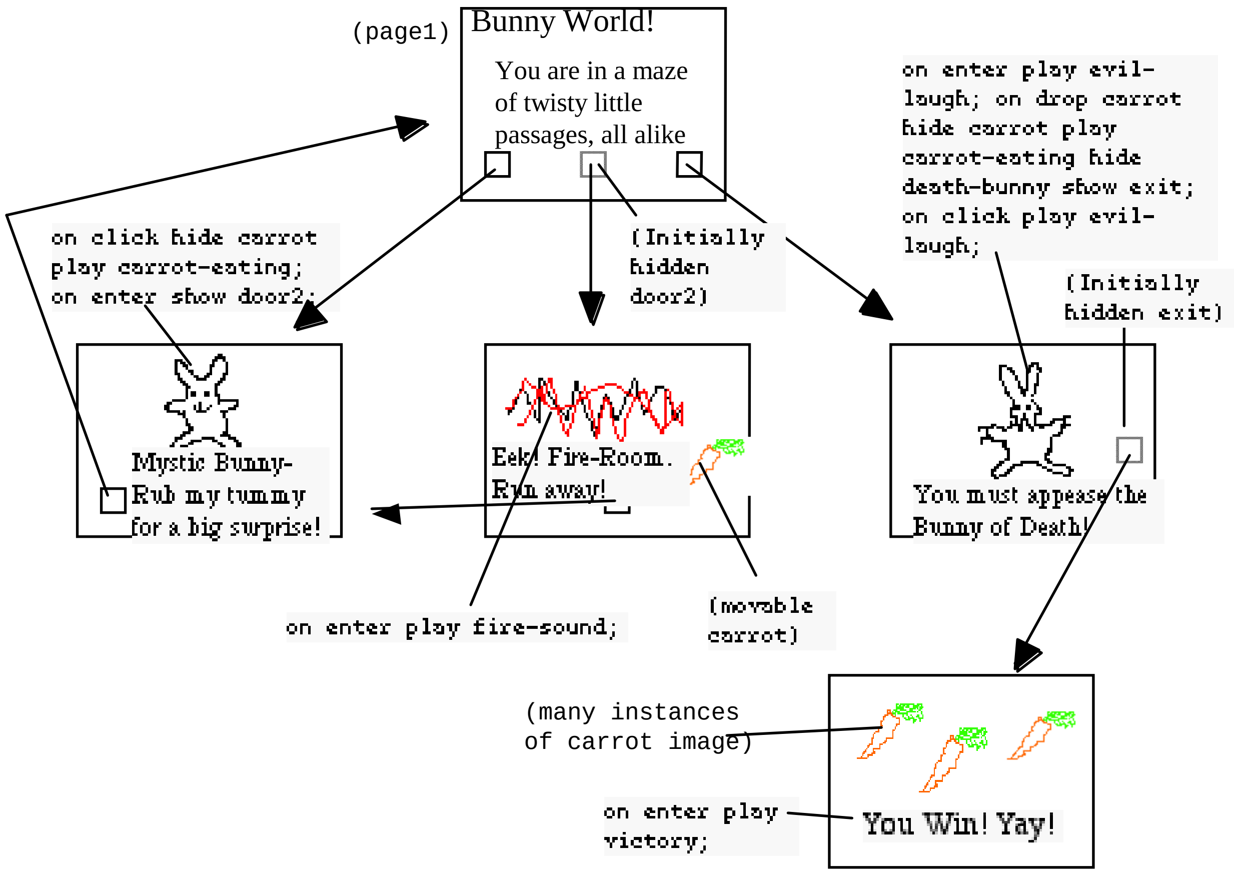
Movable shapes can be dragged around on the page.

Movable shapes can be dragged between the possessions area and the current page.

Movable shapes can be dragged and dropped onto other shapes to trigger their "`on drop`" clauses. Receiving shapes should give visual feedback with a green border during drag-and-drop mouse tracking to indicate that they have an "on drop" clause for the dropped object. For example, if the bunny shape has an on-drop clause for the carrot, the bunny should draw the green rectangle during mouse tracking when the carrot is dragged over the bunny. If a shape is dropped onto a shape that does not have a matching on-drop clause, the dropped shape should "snap back" to where it started— either on the page or in the possession area.

Bunny World Example

Here's the original Bunny World adventure that was the seed idea for this whole project. Each rectangle represents a page with some shapes in it. Each little rectangle represents a little shape with a "`on click goto <dest>`" clause where <dest> is the name of the page at the end of the arrow.


(page1)
Bunny World!
You are in a maze of twisty little passages, all alike
on enter play evil-laugh; on drop carrot hide carrot play carrot-eating hide death-bunny show exit; on click play evil-laugh;
on click hide carrot play carrot-eating; on enter show door2;
(Initially hidden door2)
(Initially hidden exit)
Mystic Bunny- Rub my tummy for a big surprise!
Eek! Fire-Room. Run away!
You must appease the Bunny of Death!
on enter play fire-sound;
(movable carrot)
(many instances of carrot image)
on enter play victory;
You Win! Yay!

The solution path is:

Take the leftmost door to visit the Mystic Bunny. Rub the tummy not, you shall! Visiting the Mystic Bunny causes door 2 to appear. Go back to the beginning and follow door2. Drag the carrot to the possessions area. Go back past the Mystic Bunny (not rubbing his tummy) to the start. Now you can face the Bunny Of Death (note the huge fangs). Drag the carrot to the Bunny Of Death. The Bunny Of Death vanishes, the exit door appears, and you can go to the end page. Yay!

> Haven't we all been tempted by our personal Mystic Bunny? Must we not each face a Bunny of Death?”

Part 2 — Bunny World Features

Given the rules that govern Bunny World, this section introduces the most important features for a functional Bunny World editor. The way the game appears and operates while playing is pretty well defined. On the other hand, the game editor side of the problem has many possible solutions. This section lists the features that the editor should support.

Goal

The goal for the project is build a program that can create and play graphical adventure games. The solution should strive to be simple but effective. There's no way for the program to include every possible feature, so it should concentrate on a core of consistent, useful features which best allow an author to build adventures such as the "Bunny World" on the previous page. The feature set may be basic, but what is present should be solid and usable. In particular, features should not crash or the documentation (described below) should warn us. As a practical matter, this means most of the functionality needs to be really done at least 10 hours before the final due hour to allow time to hunt down and smooth out any erratic areas. Sometimes projects are turned in where some basic feature, like file opening, was broken by a late feature-add, but nobody noticed. This is also why modularity is important -- as much as possible, adding features in one area should be insulated from all the other areas.

Priorities...

1. Reliable

2. Core features necessary to construct and play the example Bunny World document

3 Extra features

We prefer a program with 50 100% done features to a program with 60 90% done features, and that will be reflected in the grading criteria. For full credit, a feature should work correctly in all reasonable circumstances and without drawing glitches or exceptions. When grading, we look at the total number features shown in the program, and for each feature, we evaluate how useful it is for the user, how much work it is to implement, and if the feature is buggy. We give more points for a feature that is implemented perfectly compared to one that has great potential, but is buggy in some cases. Things we regard as bugs include: the GUI gets in an illogical or non-functional state, the drawing is wrong, the data is garbled, exceptions appear.

Target User

The official target user for the graphical adventure editor is a technically savvy user who is familiar with the rules, language, and structure of the graphical adventure format as described in section 1 above. The user will not be familiar with your editor's particular GUI.

Core Features

The following core features represent the basic functionality the editor needs to create and play Bunny World documents. From a grading point of view, the core level of functionality is very respectable work. A project that does all the core features perfectly is around a solid "B" (see the Advanced features in the next section).

You may implement commands using either buttons or menus (menus take up less screen space). The main page, possessions, inspector and catalog should all be in one big window in the Standard Swing style with the content in the middle and the controls around the perimeter.

- There should be an about box window available that lists the team name and its members and anything else you feel like including. Feel free to have a creative about box.

- The window shows the current page, possessions area, and most of the controls. The window should have its own little File controls with "*Open*, *Save*, *Save As*, and *Quit*" options, and should respect the dirty bit correctly. The Open command should effectively close the current document state (prompting to save if it is dirty), and then choose a new document to open into the window.

- There should be a way to create, name, delete, and see pages. Newly created pages should automatically be assigned names. The first page created should always get the special name "`page1`". Subsequent pages should get the names "`page2`" , "`page3`", etc. There should be visual feedback of the name of the currently displayed page. There should be a way to edit the name of a page (except `page1`). There should be a way to delete the current page (except `page1`), which deletes the shapes on that page (though not the resources they refer to). The program does not need to detect errors in page names such as names that conflict or names that contain whitespace or semicolons.

- There should be *Next* and *Previous* page commands that switch the currently displayed page in the window in accordance with the chronological order in which they were created — e.g. *Next* from `page2` should show `page3`. *Next*/*Prev* do not need to wrap around the page ordering.

- There should be a way to add, name, see, edit, and delete shapes in the current page. Newly created shapes should automatically be assigned globally unique names "`shape1`", "`shape2`", and so on. There should be an "inspector" area in the main window which displays and edits the state of the currently selected shape. When no shape is selected, the inspector should be disabled. The state of the selected shape includes: its geometry (4 ints), name, movable, visible, image-name, text, font, bold, italic, and script text. The mouse should be able to move the shape and resize it by its knobs. The inspector should synch in realtime as the shape changes. It should also be possible to change the geometry of the shape by editing one of the four 4 ints and hitting return. There should be a combo-box control that shows and edits the current image-name for the shape, or "-none-". It's acceptable to use buttons for commands (as we did with Whiteboard).

- There should be a way to add image and sound resources to the game. The program should use the standard file chooser to select a resource to be imported into the current document. When it is first added, each resource will have a full file path, such as `/usr/class/cs108/hw/bunny/resource/carrot.gif`. Resources should use the last, filename part of their path as their initial name in the editor — such as `carrot.gif` for the example above. We will not worry about the case of loading images with the same filename from different directories. (The handling of resources changes when the document is saved below.)

- Each image and sound resource should be loaded into memory just once in the data model, so for example, there's just one image object for the `carrot.gif` image. The higher levels of the program should share that one copy, getting a pointer to it from the data model by name. Resources should load when they are actually used -- otherwise there's an annoying pause when first opening the document.

- There should be a way to save and restore the entire document. Each document will be saved as a directory with files in it. When saving, the editor should use a standard chooser dialog to prompt the user for the name of a directory. A directory with the given name should be created (if necessary), and the state of the document should be stored in that directory. The directory should contain two things: (a) every resource file used in the document, and (b) a single `index.xml` file containing the document state (detailed below).

- Each resource should be stored in a separate file. At save time, the raw bytes of each unsaved resource file should be copied from its original path `/usr/class/cs108/hw/bunny/resource/carrot.gif` to the chosen document directory `~nick/bunny/mydoc/carrot.gif`. Use `BufferedInputStream` and

`BufferedOutputStream` to copy the bytes over, 4000 bytes per read or write. (Do not use the "reader" and "writer" classes -- those are for text files, and don't copy the data over 1 byte at a time, that's too slow.) After the resource has been saved, the data model should know that a request for `carrot.gif` should be served from the document directory, not the original resource path. Before a resource is saved, the program uses the original resource path. After the document has been saved, the program prefixes the document path to the resource name. In this way, it should be possible to move a document directory to somewhere else in the file system or to another computer, open it, and have everything work. The program needs to have a dynamic notion of "the document directory:" which is prefixed to find the document files. Use the correct separator char from the `File` class so your document can be moved across OSes and still work.

- The "index.xml" format should store all the document page and shape data with some XML format of your choosing. You probably want to use the xml reader/writer machinery. The XML dump of your own document can also be helpful for debugging/verification since it basically shows the whole state of the document in a human readable form. Note that no full path is ever stored here.

- When launched, the program should check for a single directory-path argument on the command line. If present, the program should open that document. The program should accept the directory path with or without a path-separator-char as its last char and do the right thing. Given the name of the directory, the program can read the index.xml and other files from inside that directory.

- Unlike Whiteboard, the "shape" and "model" objects should not have a simple 1-1 relationship. Every shape should have a model, but some models may not have shapes at some times. This is simply because only one page is on screen at a time. When opening a file, create all the necessary model objects. Only create shape (view) objects for the models on the currently displayed page. A typical document might have 100 shape models total, only 5 of which are on screen at any one time. When switching pages, delete/release the shapes no longer shown, and set up shapes for the new page. If any object is being deleted, remember to have it un-register if it is listening to something. Only creating the shapes for the currently shown page will allow the program to pop up quickly when first opening a document.

- There should be a "catalog" interface that presents a list of all the elements in the game by name and category (page, shape, image, sound). The pages and their shapes should be displayed in a 2-level hierarchy -- the name of each shape under the name of its page. Selecting a shape in the main window should select that shape in the catalog. Likewise, double clicking a shape in the catalog should switch to that page and select that shape in the main window.
  The resources should be displayed by name in a separate catalog with separate sections for images and sounds. At a minimum, the names of the available resources should be visible, and there should a command to delete a selected resource. The pages and resource may be combined into one big catalog or may be left as two catalogs. Your catalog-redraw machinery should be smart enough to not rebuild the whole catalog on trivial shape changes, such as moving them around the page or editing their script.

- There should be a "play mode" to play the game. Going into play mode should first back up to memory the current state of all the shapes and pages but not resources (the resources do not need to be backed up since play never changes them -- this keeps the backup cheap.) Editing and catalog operations should be disabled in play mode. There should be a way to go back to "edit mode" from play mode which should discard the changes made in play mode and revert to the backed up state (to avoid generating unnecessary "do you want to save" alerts, restore the state of the dirty bit from backup as well). Switching from play mode to edit mode should not change which page is currently displayed. There should also be a "test play mode" that is just like play mode, except it stays on the currently displayed page -- handy for debugging bunny script. The program does not need to support file operations for games during play. The official policy for Bunny World is to leave all error checking until as late as possible. The program can leave consistency checks of the game (references to non-existent elements, syntactically incorrect scripts, multiple objects with the same name...) until play time. In the simplest case, play time errors can just bring up an alert that something went wrong. Most importantly, the program should not misbehave or throw exceptions.

- There should be an approximately two page user document written in HTML that summarizes the features of the program for a normal user, in a file named: `UserDocumentation.html`. You may assume that the user is familiar with bunny world concepts, and just explain how to use your program. Part of writing a real program is having some docs. Trying to write docs can also help you notice if some important feature is hard to describe how to use.

  The docs can be pretty brief. The documentation can be frank about what works and what doesn't, to help guide the TA in grading. Please list what features work (plus any quirks or constraints we should know while testing) and any extra features we should check out. Please try to list the features in approximately the same order as "Core Features" list above to smooth our bookkeeping while grading.

- It's acceptable for the implementation to use "simple" algorithms such as a linear search in the data model, so long as the UI is reasonably snappy. As a practical matter, BunnyWorld is not very demanding for performance, so it's a good strategy to engineer for simplicity, moduarlity, and reliability and not worry about performance too much. Modularity and avoiding bugs will be more important than performance -- a very typical hierarchy of worries for a large, modern project.

- Whiteboard did some simple networking, however we do not recommend you try to add networking to Bunny World. Networking is notoriously hard to get completely debugged, and even without networking, Bunny World is plenty hard enough to chew up all the time we have available. The main lesson of the networking in Whiteboard was using MVC to keep things separate -- that theme remains very important in BunnyWorld. Keep things modular so they can be worked on by different people at the same time.

Advanced Features

These are features that add noticeably to the usability of the program. An above-average project should implement some of these. None of these features are as important as solid implementations of the core features above. Some are easier than others. For grading, we will consider both how much usefulness the feature adds and how difficult it is to implement. Feel free to add any feature you think is useful— this is just the list of things I've been able to think of. Please indicate any advanced features or other niceties of your program in your README so we don't miss them. You may also wish to add elements to your the base Bunny World document to demonstrate any advanced features.

- The *Save* command could enable/disable correctly depending on the state of the dirty bit. The play mode / edit mode controls could enable/disable correctly depending on the mode. A polished GUI program should get details like this right.

- The shape could have a "use image bounds" command that resized the shape to the natural size of its image -- the size where the image is not scaled at all when displayed.

- Undo support for basic changes to shapes: moving, renaming, changing the script, etc. The MVC structure makes undo easier. Undo should not have weird constraints such as "it only works until you change the selection". The depth of the undo can be shallow, but it should be simple to use. If you implement undo, please also submit a one-page or shorter "`undo-impl.txt`" file that outlines your undo strategy. We will give some preference to undo implementations that (a) use the built-in Swing undo machinery, and (b) represent changes in memory efficient ways

- Undo support for adding/deleting shapes (harder)

- Undo support for adding/deleting pages (hardest)

- Undo support that is memory efficient -- using small amounts of memory where possible. Some sort of hybrid strategy is appropriate here: use a simple memory-wasting strategy by default, but put in a smart strategy for operations that are common or use lots of memory.

- Cut/Copy/Paste for shapes from one page to another. Ideally, this should integrate with the Swing clipboard.

- The catalog can show little miniature previews of all the images inside the catalog structure (the name should still be shown).

- The shape editor combo box can also show little image previews in the combo box (the name should still be shown).

- Page background images. The page could be given the ability to fill its frame with an image. The page could scale its image to fit its frame, or it could tile multiple copies.

- Fancy shape drawing without images. It's convenient to be able to have shapes which can draw themselves in interesting ways without requiring an image or text. To support that case, it would be nice if shapes had a few alternate appearances/line thicknesses/colors so they could show up and respond to "on click" and whatnot without requiring a image. (e.g. the doors of bunny world) It's nice if you support things beyond just what Whiteboard does. Java's "Java 2D" system supports very complex drawing.

- Extend the script language with more primitives: changing a shape to display different image, move shapes around the page, "ambient" sounds that play in the background, shapes that just wander around the world on their own for comic relief (like the bat in Hunt the Wumpus).

- Shrink into possessions. Shapes could automatically shrink and/or position them selves neatly when dragged to the possessions area. Note that a typical game will rarely have more than 5 shapes in the possessions.

- Resource deletion. Deleting resources from the in-memory data model is pretty easy. The slightly harder part is, the next time the doc is saved, getting rid of the right file in the document directory. Try not to delete all your source files when developing this feature!

- You could save an index.html alongside your index.xml. The HTML version could use <ul>, <img>, etc. to make the document state sortof show up in a browser. The HTML does not need to be readable, just writeable.

- When switching pages, you could do creative "transition" effects to show the new page -- like Powerpoint. Do this by imaging the whole new page to an offscreen image, and then use image manipulation to bring the new pixels into view.

- You could have a "structured" script editor that allowed the authoring of scripts with little or no keyboard typing. It's tough for this feature to add a lot of value since simple typing is very effective for scripts already. It may be better to put time into other features.

Goals and Strategy
Realize that there is a tradeoff between the number of features, and the quality of their implementation. The more features the better, but don't let your list of features get too far ahead of your quality of implementation, or you end up with your last few features which don't quite work, and make the whole program feel shaky -- that is a very unsatisfying state.

Attribution
If you do a Google search, and find great example code showing how to use, say, `JTree` -- that's great. Please feel free to use such resources, researching the writing of your own code. That's how modern coding is done. If you find some sort of Foo.jar that you want to use, that's ok, but for grading obviously we cannot give you credit for **creating** Foo.jar, just **using** it. Therefore, in your README, please spell out any non-standard resources you are using, and explain what we need to do to compile beyond a simple "`javac *.java`". On the whole, the Bunny World and the standard JDK are large and extensible enough that there is not a pressing need to add external resources.

Deliverable 1 — Early Demo
The early demo is due 8 days week before the final deadline — midnight ending Tue Nov 25th (no lateness permitted). This is during your break, but you have plenty of time to finish it before you go on break. The README format for a group submission just has multiple user lines, like this...

```
user: jsmith
user: kjones
user: ksauze
```

The credit for the early demo is simple all-or-nothing. If the demo mostly works, you get the credit. The demo should run when we invoke the `main()` in the "Bunny" class. Because these are the easiest points of the whole project, the teams should aim to have this functionality done a day early to allow for the inevitable "surprises".

Basically, the early demo should demonstrate a rudimentary page/shape view: There should be a single page. The page should support the creation, moving, and resizing of shapes. There should be a way of adding images into the document. Mouse gestures should be able to move and resize shapes. There should be a way to tell a shape to display a particular image (combo box is best, text field is ok). There should be a way to tell the shape to display some text. That's all that is required, although your team will probably want to be farther along at this point. It's fine if your early demo has other features but which are not functional or stable or if the implementation is a little rough. We will not go out of our way to try to crash the program. Here's a few of the things which do not need to work: the full shape inspector, saving, script code, sounds, play mode, page and shape deletion, multiple pages, and the catalog.

Deliverable 2 — Final Project
Due midnight, please turn in a directory containing all of your Bunny World materials described below.

A README telling us who's in the group (one per line, same as the early submit) and anything else we should know about using the project. The listing of specific program features should be in UserDocumentation.html. The deadline is serious -- there's a late penalty something like (10+num_minutes) points for late projects.

It's best if your code is portable so that it can work on any sort of machine -- Windows, elaine, Mac. The most common source of such problems is at the File path level -- assuming File paths begin with C: for example. Do a little testing on an myth or pod: editing, opening BunnyDoc, playing, etc.. You should also check your BunnyDoc on the myth or pods to make sure your doc format is really portable (note that sound does not work remotely over XWindows, so don't worry about that). For grading, we will start with the myth/pod type machines, but have other OS's available for emergencies.

All your sources should be ready to compile. We will run the program with the `main()` of the "Bunny" class. If we need to do something particular beyond a plain "`javac *.java`" to compile or run, it should be explained in the README.

There should be a built BunnyDoc directory/document with at least all of the structure shown in the example on page 5. We will open the document using the command line argument "BunnyDoc" on your Bunny main(). We use this document in part of our testing, and as a de-facto example that your program is fairly functional. Do not omit this part of the submission if at all possible, as it's an entire category in the grading criteria. If you are only partly able to construct BunnyDoc, do so and explain the situation in your README.

User Documentation. 1-2 pages of very brief documentation in HTML format for the target user explaining how to use your program.

Finally, I would also like by Fri after the due date, a short email to oj@cs from each team member with the subject "bunny analysis" with a very brief description of how the work went. (If you forget to send this in for a while -- that's ok, just send it in late.) The mail should list each team member with a very short description of what they did. For the vast majority of teams, everyone will have contributed more or less equally, and the analysis will not affect anything. For teams where there is a radical disparity, it may make some difference in the grading.

You have arrived -- completing a significant project with latest the technology under a hard deadline. Take a deep breath, and enjoy a well earned rest.

Deliverable 3 — Demos / In-Person Grading

We will have blocks of demo-times on Saturday Dec 6th where teams demo their project for the grader. We will have online signups beforehand for the demo times. The grading is all based on the work turned in by the deadline, but the demo gives each team a chance to make sure the grader understands how the program works, and provides a degree of psychological closure on the whole episode. Not all team members need to attend the demo.

Bunny Implementation Ideas

This section just has some ideas to help your team get started thinking about Bunny World. It also echoes the lecture material outlining a plan of how to deal with a team project like this. The previous section defines what Bunny World must do. This section is just suggestions.

Problem - Design - Implementation

You can "backsolve" for a design by first thinking about the most common problem "use case" scenarios the program must solve, and backsolving for the classes and capabilities necessary to solve those problems. It's useful to have a set of core use-cases identified right from the start -- if a design idea appears to work for the core use cases, it's good enough to get started. Here are a few of the core use case the Bunny World editor must solve...

Core Use Cases

- Goto a page. Retrieve a page and its shapes from the data model and get them on screen. Create the page and shapes visually. This comes up during editing and playing. The page panel can stay where it is actually. The trick is to remove the shapes from the old page and add the shapes of the new page -- thus the illusion is that you have changed pages.

- Shape display -- when switching to a page, the shapes of the new page should appear on screen.

- Shape edit. Message changes to the data model, and there is some notification mechanism to update things that depend on the model.

- Page and shape creation. Affects the current view and the data model.

- Page and shape deletion. Affects the current view and the data model.

- Catalog Display. The Catalog needs to be able to list all the page, shape, and resource entities. Clicks in the Catalog affects the current page. The catalog will need to listen for data model changes.

- Resource adding — store in the data model.

- Play Mode. Needs to back up the page and shape state. Requires retrieving and interpreting script text from the shape data model.

- File save/open of the whole data model.

Initial Design

With a basic class decomposition design, teams can divide up and start parallel development and testing of classes independently. Building and testing components independently at first allows the team to make progress in parallel.

1. Build an overall design. Optimize the design for **modularity** and **simplicity** (in other words, not generality). You need the modularity to best work as a team. Keeping things separate from each other allows the best parallel progress and integration. You design differently than you would for a one-person project. You need simplicity since there is a hard deadline with a well defined deliverable. You do not need generality for a 2.0 release 6 months in the future. The deadline is soon and immovable. Avoid doing complex pleasing designs -- do something that's simple but will work.

2. Verify and refine the class design by thinking through the actions and responsibilities of each class for the core scenarios. Don't hand-wave through this step. Have a design on

paper (no code). Actually simulate your way through how the classes interact, store things, etc. for each of the core use cases. This is the crude but effective way to identify any important issues with the class abstractions, implementations, and interactions. It's **much** easier to think it through now, identify problems, etc. before code is written. Present your design at your meeting with a staff member.

Early Team Meetings

Goals for your early team meetings...

- Create an overall design. Optimize for modularity -- keep things as separate as possible to make best use of the whole team. Pay special attention to messages between components and who is storing what and in what form. Everything messages the data model, so the external abstraction of the data model is especially important. Make sure everyone has the same understanding.

- Simulate the operation design on paper for all of the core use cases to really flesh out what messages are being sent to who. **Do not skimp on this step** – it's a great investment against later headaches. Really simulate through step by step what's going to happen for the core scenarios.

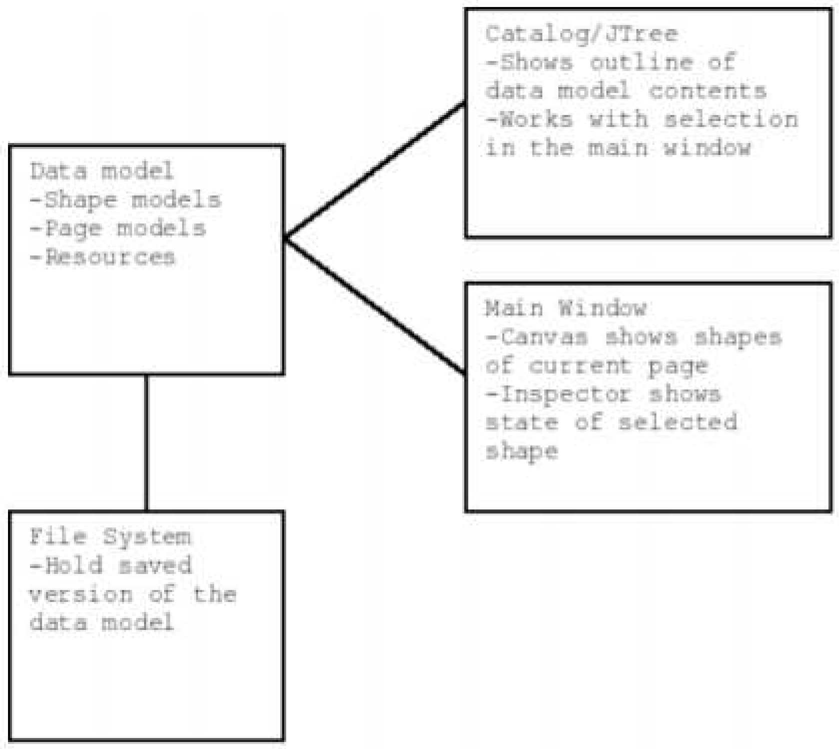
- Divide modules up for parallel progress.

- In ones and twos, people can make progress on the separate modules. Check back with the team to modify the central design as holes are found. It's hard to tell since there's not unified whole, but this is the period of maximum forward progress.

- Do integration to see how things really fit together. Expect that integration will expose new problems. Do not put off integration until right before a due date.

Modules

Here are some ideas for how to divide things up. Some tasks will turn out to be harder than others. Be flexible about moving people around to concentrate on whatever is not getting done. Many of these tasks include action items like "Research JComboBox" — one person needs to become the "local expert" on that problem. They should go read the docs, build a little test program — do whatever they need to do so they know how that built-in class works. Everyone else will need to know the role of that class in the overall program— what it sends, what it listens for — but not all the details of using it. There are many such areas in the program where one team member can invest, in parallel, in becoming the local expert for that class.



1 — Data Model

The data model needs to store all the current state -- page models, shape models, resources. There also needs to be a scheme for other parts of the program to be notified about changes to the model -- using listeners or some more primitive scheme. Everyone on the team will need some understanding of the data model since it touches everything. Whoever implements the data model can implement the file-save operations later on. For implementation, research Java facilities for manipulating directories, files, etc. The data model is a good place for client-oriented API design -- messages for the convenience of its clients -- use their vocabulary, meet their needs with a simple `getImage(String name)` that solves the client's problem in one step. Much of the code ends up being a client of the data model, so if the data model provides lots of client-convenience methods ... great.

Simplicity, Generality, and Modularity

You do not need to use a listener/notification scheme for everything. For example, if a model change is initiated in only one or two places, it's acceptable to manually call a fireXXX method after the change to update whatever object is using the model. Code which is simple but effective is good. On the other hand, code that adds implementation dependencies between modules is bad, since it makes it harder to edit one module without introducing bugs in another.

2 — Canvas/Shapes

The canvas and shape classes implement the pixels/view side similarly to Whiteboard. The data model needs to store much more than in Whiteboard. You should be able to create and delete shape objects as you change which page is currently shown. Changes need to propagate out of the data model correctly, so the inspector and catalog can hear about changes. Finally, the shapes need to be able to display scaled images. A shape should ask the data model for an image by name "`carrot.gif`" just when drawing. The shape should not store anything other than the name. The data model should know how to get a real image from the name as needed. There should only be one copy of the `carrot.gif` image in the data model, and the shapes that refer to it all use that one copy.

There are a couple ways to implement the possessions area. One way is for the possessions area to be a separate component from the displayed page canvas. Dragging shapes around adds and removes them from one canvas to the other. A possibly easier approach is to have just one component that has "page" and "possessions" areas, and then create the illusion that the possessions area is separate by not removing things from it when the displayed page changes.

3— Shape Inspector

This could be done in parallel with basic pages and shapes, or afterwards. The "inspector" needs to be able to show and edit state from the selected shape and the data model. Research using `JComboBox` to create the pop-up of image names. Research using `JTextField` to preset and edit the text state. It's ok if the user has to type return to commit what they've typed, although the slickest solution will update the state on each keystroke (see the Censor example).

4 — Catalog

Research `JTree`, so it can show state from the data model. When thinking through the use cases, make sure that the listener structure is strong enough to notify the catalog. Don't redraw the whole catalog on every little change. `JTree` is a complicated, built-in Java class. There are a lot of docs available on the Internet about `JTree`. Someone should be the `JTree` expert to research and figure out how to use it.

5 — Play Mode

The script interpreter can be done later in the project, just make sure the various classes support the operations it will need. Also the data model will need to support the "backup" operation for play mode.

Integrate and Re-Plan

- Get a realistic sense of how much progress you've made. Avoid denial.

- Make hard choices about what to do next. Some things are going great, some things are not. Re-assign people to avoid blocking on one "problem" component. Arrange things so everyone has something useful to be working on. Avoiding betting the whole project on one person getting a particular thing done in the last couple days, especially if they've had problems with it in the early stages.

Misc Suggestions

- Consider using Java interfaces to express the interface that each class exposes to the others.

- On Unix, use the "gaintool" to adjust the sound volume

- Use the `Toolkit` class to implement beeping.

- Use the `JOptionPane` class to bring up little alert dialogs.

- There is some sound and image example code in the hw directory.

- The ability to dump out the XML state of the whole data model (or for just one shape) can be quite handy for debugging ("transparency"). Have a utility button for debugging that prints all the XML state to standard out or to a file.

- Your team should use CVS or some other source control system.

- Swing has built-in drag-n-drop support, however it has a reputation of being a little complicated, so it is probably not worth using for the simple dragging operations needed in Bunny World.