

Bunny World

due 5:00 p.m. Thu Dec 2nd

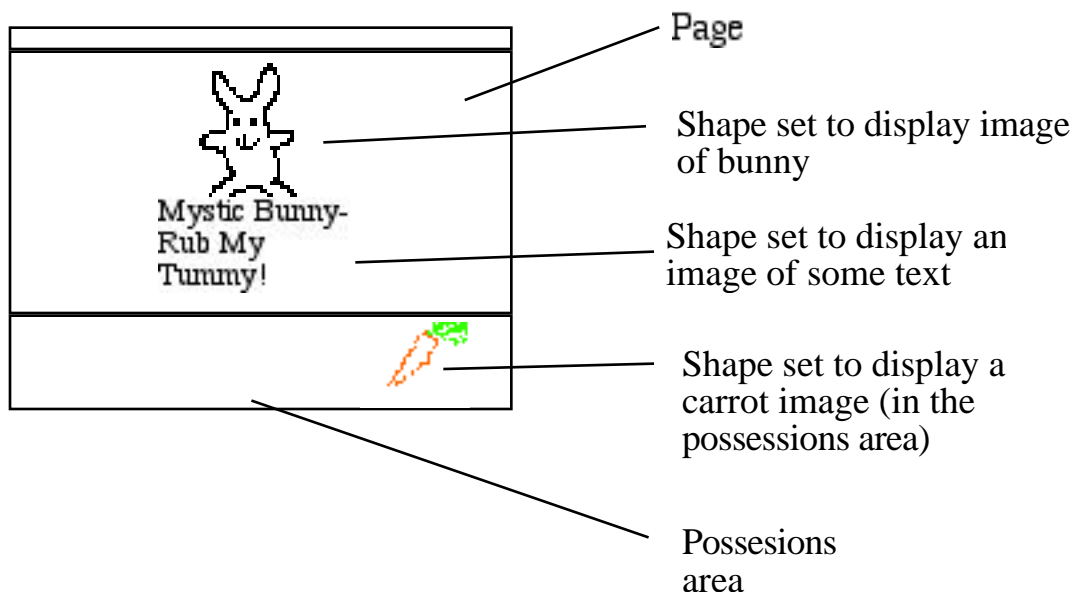
The Final Project gives you a chance to use the latest technology to solve a significant problem with a real deadline. If and when your Bunny World comes together, it's a moment of well earned joy.

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.

A separate Bunny World Ideas handout gives suggested Human Interface (HI) and implementation ideas. Those will just be suggestions— any editor which 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 which 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.



The basic elements of the game are...

1) Pages

A page is a 350 x 250 rectangular area that can contain a number of "shape" objects. The world contains many pages. Only one page and its contents are visible at a time. Certain actions within the game will switch the game to show a different page and its contents. 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 which identifies it. For all the operations below that involve names or other text elements— nothing should 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 MiniDraw, each shape has a bounding rectangle which can be moved and resized. The shape draws itself simply as a light gray rectangle.

Each shape can refer to the name of one image which it draws. The shape should draw the image scaled to fill the shape's bounds. If a shape does not have an image or the image cannot be loaded, then the shape should draw a light gray rectangle instead. The shape does not own "the picture, it just uses it by name — multiple shapes can use the same picture.

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 where the user may drag movable shapes during play mode. The possessions area operates like a permanent little page where shapes may go. During the play of the game, as the player moves from page to page, the possessions area provides a way for the user to carry shapes 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 bottom part of a shape is clipped off when it is in the possessions area.

4) Resources

The game document contains image and sound resources. The user adds the resources to the document by selecting them using a file chooser on the file system. The document should make its own copy of the resource -- it should not depend on the continued existence of the original. The shapes refer to the resources by name for drawing and playing purposes. Each shape stores the name of one image which it uses 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. (We will provide some help to get you started with the built-in image and sound classes.)

5) Shape Script

Every shape has a block of "script" text which programs how it 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 each 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 runtime to control what happens. For example the clause..."on click play hoo-aah goto home-page;" within a shape means that when that shape gets clicked, it plays the sound named "hoo-aah" and then the game should switch to display the page named "home-page".

Script Actions

There are five script primitives which 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. The script should contain at most one on-click clause.

- 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.

- 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.

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

The game starts by showing the page named "page1".

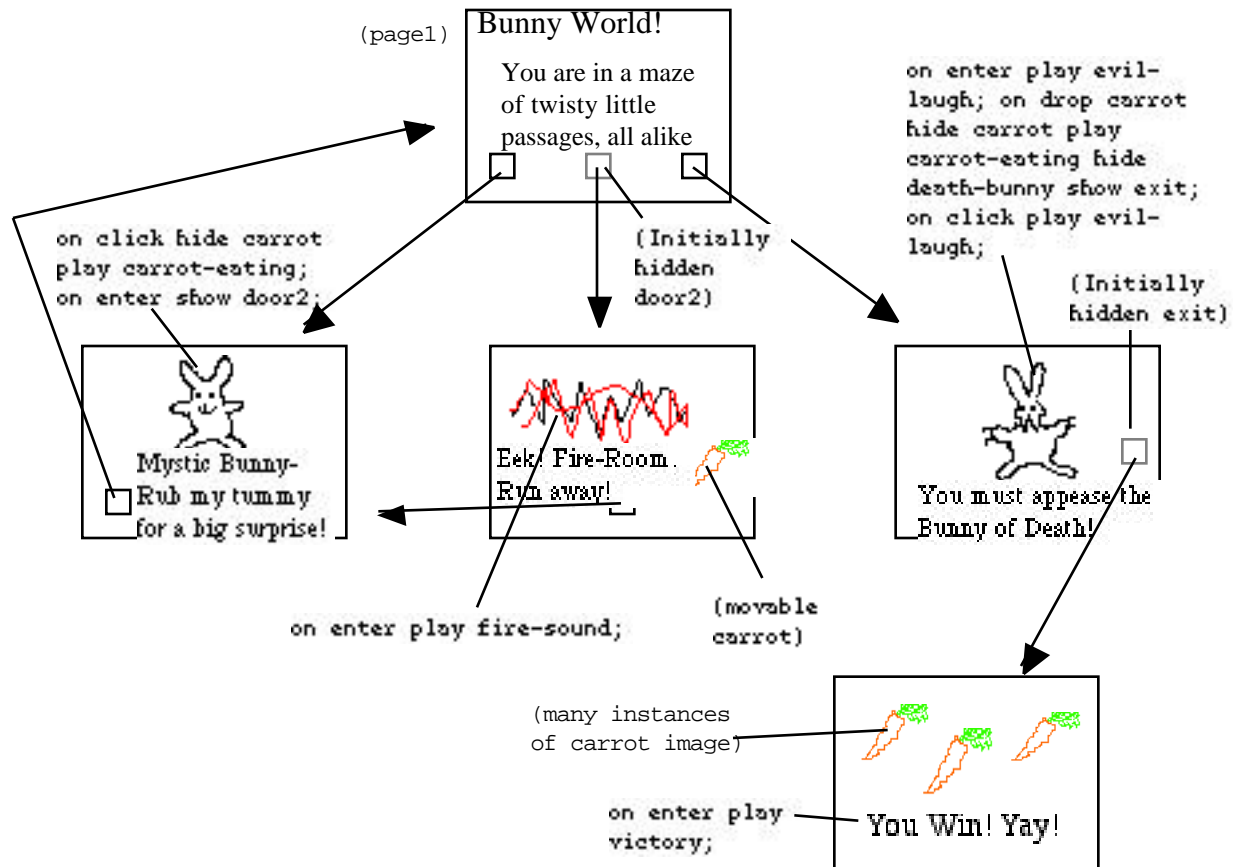
Movable shapes can be dragged around on the page.

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 during drag-and-drop mouse tracking to indicate whether they have an "on drop" clause for the dropped object. If the receiving object does not accept the dropped object, then the dropped object should snap back to where it started— either on the page or in the possession area.

Bunny World Example

Here's an example adventure which demonstrates the basic structure of the game and its code. 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.



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!

“Bunny World — an anthem for our times. 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 which 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 which the editor should support. (For remote students working on their own, there will be a slimmed down version of these requirements — see the web page.)

Goal

The goal for the project is build a program which 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 page 5. 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 done at least a couple days before the final due date to allow time to hunt down and smooth out any erratic areas.

Priorities...

1. Reliable
2. Core features necessary to construct Bunny World document
- 3 Extra features

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 the editor's particular HI.

Core Features

The following core features represent the basic functionality the editor needs to create and play Bunny World documents. You may implement commands using either buttons or menus. You may put all the controls in the main window, or you may put some in separate windows.

- There should be an about box listing the team name and its members and anything else you feel like including. Feel free to have a creative about box.
- There should be a main window that shows the current page and most of the controls. The program should maintain a single document represented, at least, by a single window which shows the "current" page, the possessions area, and the controls. We're going to ignore the inner-frame classes, since having multiple

documents open at once doesn't seem every important for Bunny World. Instead, we'll go with a single window that represents the currently open document. The main window should have its own little File menu with "Open, Save, Save As, and Quit" options. 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. For simplicity, the pages may all be a fixed size of 350x250. 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. The window should include the name of the current page in its title. 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 which conflict or names which contain whitespace or semicolons.
- There should be "Next" and "Previous" page commands should switch between pages 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 unique names like shape1, shape2, etc.. There should be controls in the main window which display and edit the state of the currently selected shape. When no shape is selected, they should display nothing. In addition to mouse gestures to move and resize each shape, there should be controls to edit all of the shape state its name, its visible and movable booleans, and its script text. There should also be 4 small text fields showing the (x, y, width, height) of the currently selected shape. Entering a valid number of one of these fields should change the state of the selected shape. There should be a "pop-up" control that shows and edits the current image for the shape, or "-none-". It's acceptable to use buttons for all of the commands (as we did with MiniDraw). (If text-focus support in Swing were better, we could use, for example, the delete key to delete shapes, but text-focus support is poor, so we won't bother.)
- 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. Resources should use the name of the file that produced them, such as "bunny.gif", as their name in Bunny World.

- There should be a "catalog" interface which 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 be a command to delete a selected resource.
- There should be a "play mode" to play the game. Going into play mode should first back up in memory the current state of all the shapes and pages (note that the resources do not need to be backed up, which 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). 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 a way to save and restore the entire document. When saved, the game should have its own copy of all its resources. File operations do not need to be supported during play mode. 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, and the state of the document should be stored in the directory. Each resource should be stored in a separate file. The rest of the state may be stored in a custom format file in the directory as we did for MiniDraw. When saving, the program should first remove the contents of the directory if it exists, and then write the state into the directory from scratch (this way, resources which are deleted in memory are deleted on disk). 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 save directory should also contain an index.html that shows: all the images, and using nested ``'s, all the pages and their shapes. Each shape, in cryptic text form, should show all of its state. This is mainly just a debugging feature. You may find it useful that

you can essentially dump all of the current document state to text for examination — this is sometimes called "transparency".

- There should be an approximately two page user document written in HTML that summarizes the features of the program.

Advanced Features

These are features which 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 controls in the main window could enable/disable correctly depending on whether a shape is selected. Similarly, 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. (Swing has built support for undo, but it's complex.)
- Undo support for adding/deleting shapes (harder)
- Undo support for adding/deleting pages (hardest)
- Cut/Copy/Paste for shapes from one page to another (cool)
- 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.
- Shape drawing without images. It's convenient to be able to have shapes which can draw themselves in some way without requiring an image. 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)

- **Text shape.** Often, the author will want to include a shape which displays some text. The core features allow this in an inconvenient way — the author can create some text in a draw program, and then import it as an image. It would be more convenient if the shapes had some capability to display a block of text on their own. The text could be drawn on top of the shape's image. The text could automatically pick the right font size so that it more or less fills up the bounds rectangle (this makes an awesome looking effect as you drag the knobs around and the text resizes on the fly). There could be controls to edit the font, style, etc. of the text.
- **Clipboard resources.** It would be nice if the user could paste image or sound resources into the document as an alternative to using the file system. Historically, the clipboard interface with the system has been a little sketchy, but it may now be solid enough to support this feature. (Drag 'n Drop is an alternative to the clipboard, but the system drag 'n drop support is also of unknown quality.)
- **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 Hump The Wumpus).
- **Shrink into possessions.** Shapes could automatically shrink and/or position them selves neatly when dragged to the possessions area.
- **The resources should definitely be saved as separate files.** However, for the shape, page, etc. file info, you could use XML instead of serialization. You should use some of the existing XML Java libraries. See <http://developerlife.com/>. This is a non-trivial feature, but if you want an excuse to learn XML, well here's your chance. If the XML support is working, your project does not need to implement the HTML index.html feature.
- **When switching pages, you could do cheesy "transition" effects to show the new page.** Do this by imaging the whole new page to an offscreen image, and then use image manipulation to bring the new pixels into view.

Strategy

There are two paths for a high quality Bunny World. Some projects excel by implementing a relatively small number of features, but doing an absolutely solid job of it: everything works, looks good without drawing glitches, no exceptions etc... Other projects add several features, but without having completely solid implementations of many features. Realize that there is a tradeoff between the number of features, and the quality of their implementation. Don't let your list of features get too far ahead of your quality of implementation, or you end up with something with a lot of features which doesn't work — this is an unsatisfying state to get in. In the grading we try to recognize the pluses and

minuses of both the number of features and their quality, but there is a preference for quality. There are points which are earned by having a "solid" feel — a clean visual look and no blinking, visual glitches, exceptions, or lockups.

Deliverables 1 — Meet With a TA

Every team needs to choose a team name like "Mystic Bunnies," "Nerd Meltdown," or the ever popular "Show Me The Bunny." The team name may not include the phrase "bunny world". Every team needs to meet with a staffer on Mon, Tues, or Wed. Signups will go up on the web Sat afternoon. Each team should take about 20 minutes (eg 11:00, 11:20, 11:40.). First priority goes to people who sign up, then first come first serve. Bring your "design diagram" which shows how the modules are arranged, what features they supply to each other, and who is working on what when.

Deliverables 2 — Early Demo

The early demo is due about 2/3's through the project period — midnight ending Wed Nov 24th (no lateness permitted). Please include a README which gives the team name, who's in the group, and their emails

Caffeine Extravaganza - Nick Parlante, Tasha Yar, Yaphet Kotto
nick.parlante@cs, yar@cs, yaphet@cs

As usual, you should mention any other things we should know in the README. The early demo is worth approximately 10% of the total points for the project. The points are coarse all-or-nothing. If the demo pretty much works, you get the points. 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 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 moving and resizing shapes. There should be a way to tell a shape to display a particular image. 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 entirely functional or stable. We will not go out of our way to try to crash the program outside of the few required features.

Here's a few of the things which do not need to work: Saving does not need to work. Script code does not need to work. Sounds do not need to work. The full suite of shape editing controls do not need to work. Play mode does not need to work. Page and shape deletion do not need to work. Multiple pages do not need to work. The catalog does not need to work.

Deliverables 3 — Final Project

By Thu Dec 2nd, 5:00 p.m., please turn in a directory containing all of your Bunny World materials described below.

A Readme using the above format telling us the group name, who's in the group, their emails, and anything else we should know. Comments about specific program features should be in the user documentation. If we need to do something with the classpath to run your, please say so in the Readme.

All your sources should be ready to compile. We will run the program with the main() of the Bunny class.

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. If you are only partly able to construct BunnyDoc, please explain the situation in your Readme.

User Documentation. 2 pages of documentation in HTML format for the target user explaining how to use your program. The documentation can be frank about what works and what doesn't. We will read through your documentation at the same time we start play testing our way through all your features. Please list what features work (plus any quirks 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.

Finally, I would also like by Saturday the 4th, a short email to nick.parlante@cs from each group member with the subject "group analysis <groupname>" analyzing how the work went. The mail should give the group name, and list each member with a short description of what they did. For most groups, everyone will have contributed more or less, and the analysis will not affect anything. For groups where there is a radical disparity, it may make some difference in the grading.

You have arrived. Take a deep breath, and enjoy a well earned rest.