THE 6.001 ADVENTURE GAME: NAVIGATING THE JUNGLES OF MIT

Team 18: Kyle Ingols, Aileen Tang, Lawrence Wang

Abstract:

For this project, we built a graphical version of the 6.001 text-based adventure game. We wrote a Java version of the original Scheme game so we could interface it with runtime image generation and display. On the graphics side, the objects and people were modeled using Alias Wavefront. We obtained the scene models by using Canoma to generate 3-D models from photographs of the actual locale and to texture map the photographs onto the 3-D geometries. Alias objects were then incorporated into specific locations in each scene model and ray traced using a light source that closely resembles that of the original photograph. We then used an image diff technique to enable fast reconstruction of all possible combinations of people and objects per scene, without having to prerender all possible combinations -- allowing real time image generation as the adventure game is being played. We encountered obstacles with integrating outputs of one tool with another and learned valuable lessons in the process about realistic tools assessment and effective project planning and management. Overall, we learned to search for tools that satisfied our needs, learned to use these tools and to find ways to work around them if they do not work, and produced as a product a graphically enhanced adventure game that is more interesting and fun to play.

Table of Contents

Introduction

A. Background

Our project stems from the 6.001 Adventure Game problem set, which acclimates students to object-oriented programming by playing a simple text-based adventure game. Everything within the game is modeled as an object. There are three main classes of objects: places, people, and things. Places can be various corners of the MIT campus, and people can be either autonomous-persons (robots that define their own actions at each clock tick), the single avatar (the protagonist character that the player controls), and trolls (the bad guys that raid campus and attack other people). Things are non-people objects that either exist in places or in people's possession.

Clock ticks define the time in the world of our adventure game. At each clock tick, each autonomous person and troll randomly picks a possible action and performs it. Some example actions include EAT, SLEEP, MOVE-TO another place, or HIT a victim (if the person is a troll). On the other hand, what the avatar does at each clock it is controlled by the player. The goal of the game is to navigate the halls and buildings of MIT until the player finds and obtains the thing named 'diploma.

B. Motivation

While the adventure game serves the purpose of introducing object oriented programming to 6.001 students, it is also a lot of fun to play. The objects' actions in the game produce very interesting (and

often hilarious) interactions. And since MITers are all in fact playing the game in real life (wondering around the institute sleeping, eating, suffering, and hoping to snatch the diploma before getting killed by trolls like the registrar), we can actually relate to the otherwise amusingly absurd situations generated by the game. Therefore, we thought it would be fun to create a graphical version of the original text-based game. The addition of graphics gives an additional dimension to the game and makes it not only conceptually interesting but also visually fun.

The graphical game is in storybook style. Each clock tick produces a rendered scene based on the state of the adventure game world at that moment. The player sees a scene of where the avatar is currently located, along with other objects and people located in the same place as the avatar.

C. Computer Graphics Techniques

Computer graphics help bring text to life, and our project enhances the visual experience of the adventure game. The style of placing cartoon-like characters and objects inside of realistic scenes of the actual MIT campus captures two unique aspects of the adventure game that makes it so much fun to play. The characters' experiences, often funny and absurd, give them a cartoon-like quality. Yet the setting and premise of the game is so familiar to the players (assumed to be MIT students) that the combination of familiarity and hyperbole makes the game interesting to play. Relevant computer graphics topics include scene and object modeling, ray tracing, and our experimental notion of image diffing (described throughout the paper).

Goals

A. Project

We begin by creating models of each PLACE in the world. (In this case the PLACES are familiar MIT locales, like 10-250 and Lobby 7.) We will also model the people and objects, which include denizens of 10-250 (Prof. Duane Boning, a sleepy student) and some objects as well (a problem set, a bottle of jolt). As a consideration of the speed/quality/storage tradeoff, each PLACE is assigned a single view angle, meaning that there is no animation of views. This allows us to pre-render a good, ray-traced view of the room and use it each time the room is entered. In addition, each PERSON and THING is assigned a non-overlapping location within each PLACE. This means that no given object will ever occlude another object.

We opted to make the quality of the presentation high by sacrificing storage space and preprocessing time. Before ever playing the game, we create a large quantity of pre-rendered ray traced images, and then combine those images at runtime to present the desired picture to the user. For lack of a better term, we call this image diffing.

We model each of our scenes (i.e. PLACES) from photographs of MIT locales and texture map the photograph directly onto the model so it has a realistic semblance to the actual place. For each scene, we first render it empty. Then, we render the PLACE containing precisely one object, take a diff of that picture and the picture of the empty place, and store the diff. We do this for every possible person and object that can potentially be in each scene. At runtime, our game engine decides which objects are in the current scene, loads up the image of the empty PLACE, and patches in the diffs of each object that is present. This will not give 100% accurate shadowing, but it allows us to get a relatively decent image compared to one that is a properly ray traced and far superior to a Gouraud-interpolated representation

of the scene rendered on the spot.

The original object-oriented adventure game is written in Scheme for the 6.001 problem set. We will first port this game to a more useable language (e.g. java) and modify its output code to both print a textual representation of the game state and interface with the graphics generator. When we pass the state of the game to the graphics generator, it fetches the necessary images and masks, performs a real-time application of the masks to the image, and displays the result to the screen.

B. Extensions

The scalability of the project allows for many extensions as time permits. For example, cut scenes to animate travel between two places could be added. It would also be interesting to include animations that show character actions (picking up items, dropping items, speaking, attacking other people, and so on). The interface to the game itself could be changed to use the mouse; moving over an object can highlight it and show its name, dragging the object to the person could TAKE the object, clicking on an exit could leave in that direction, and so on.

More simple animations would be more feasible, for example people waving their arms, a revolving object name above each object in the scene, or a short sequence of a person being attacked by a troll. In the case of the moving people, we could simply render small one- or two-second animations of the action for each scene and person, and store those animations (perhaps as a lower-quality animated GIF) instead of the normal masks. This would require the graphics engine to re-mask the background image quickly, but it is expected that the simplicity of the task will easily allow 15 or more frames a second. In the case of adding revolving text, each object's name can be rendered as a spinning animated GIF and simply inserted in the proper location, regardless of the place in question. This would be a small, non-bottleneck addition in terms of preprocessing time, running time, and storage.

Achievements

Our achievements fall into three main categories, detailed below with a discussion of what obstacles we faced and how we dealt with them.

A. User Interface/Java Game Engine

We ported the game from Scheme to Java and connected it with a front end user interface (Figure 1). Combining Java Swing light-weight components with the new features of jdk1.2 such as Drag and Drop capability, we were able to construct a decent-looking interface with good usability in a relatively short period of time. Because of the short time frame for the project, the initial focus was on function rather than features and look, so as to allow us to debug the game algorithm and graphics as soon as possible; however, many advanced features and improvements on usability have been made since then.

The interface consists of three panels: view panel, inventory panel, and control panel. The view panel displays a first-person view of the game scenes rendered in 3D. It takes up most of the space in the center of the user interface. When the avatar moves around in the virtual game environment, the view panel updates itself to show where the avatar is, what items and persons(if any) are at his/her current location.

The inventory panel shows a list of items the avatar is currently carrying, with graphical icons alongside

text display for each item. This list updates itself as new items are acquired, or old items are used up or dropped. To acquire an item, the player can either drag the item from the view panel to this panel or use the "Pick" button(see control panel). Some items can be used by double-clicking in this panel.

The control panel has all the command buttons the player needs to navigate and perform actions in the game world. Each button and its function is listed in the following table:

BUTTON	**FUNCTION which it performs**
North South East West Up Down	Move through the game world
Pick	Take a chosen object from the environment
Drop	Get rid of something you are carrying
Use	Make use of something you possess (eat it, unlock a door with it, etc.)
Sleep	Do what MIT students never have time to
Look	Examine your surroundings more closely

The porting task was considered bootstrapping and did not consume too much of our efforts. However, we did run into the difficulty of warping Java, a strongly typed, virtual machine language from Scheme, which has very little type checking and runs in an evaluator. We "hacked" Java to simulate the workings of the original game, where all of the user commands were evaluated directly as Scheme expressions directly inside the interpreter environment. To do the same thing in Java, we had to simulate the Scheme read-eval-print loop and get Class methods based on input arguments, which also include the method's parameter list and the type of each parameter. The resulting command line interface was awkward, because it required the user's knowledge of the Java method names (rather than a quoted message in Scheme, like "eat") and the types of the arguments that the method needs. However, the graphical user interface layer alleviated this burden from the user by providing buttons that abstract out the details of our method calling convention. The resulting Java engine may not demonstrate the most elegant object oriented design, but it does the job, as long as we do not use it for teaching object oriented programming in 6.001.

B. Object/Scene Modeling

We modeled the people and objects using Alias Wavefront. The design criteria for people was to come up with a general shape for people and customize it for each individual character by adding accessories such as glasses, hats, weapons, and facial features. (Two of the people are shown later in the paper.) As mentioned before, we wanted to capture the game's whimsical quality by creating cartoon-like characters and objects. The overall effect of these funny-looking characters, chubby with large hands and feet, overlaid on a texture-mapped scene creates a style of humorous exaggeration.

The scenes were modeled using Canoma, a software package that makes 3-D models from 2-D images. We took digital photographs around MIT campus and used Canoma to create 3-D geometries of the parts of each photograph that may interact with the objects we placed in them. Then Canoma texture maps the photograph onto the geometries to recreate a scene that closely resembles the original photograph. Since we did not need to create scenes that can be viewed at multiple angles, the scene modeling work was not too complicated.

However, incorporating the files exported by Canoma into Alias Wavefront was a non-trivial task. The only file format powerful enough to retain texture information and common to both Alias and Canoma is the Wavefront .OBJ format. This text-based format contains an .OBJ file, which specifies the geometry and the shaders, and an .MTL (MaTeriaLs) file, containing definitions for each shader (including the particular textures, reflectivity, emissivity, and so on).

Unfortunately Alias refused to interpret the .MTL files produced by Canoma. Searches on the imagery5 mount-point and the web at large failed to yield any examples of a good, correct .MTL file, or a summary of the file's specification. This effectively meant that we were only able to leverage Canoma's textures by methodically adding the textures to each shader in each scene, one at a time. The tedium is impressive.

What's worse is that we discovered some nasty interactions between the geometry which Canoma provided to Alias and the Alias rendering engine's efforts to texture-map the geometry. Canoma returned the scene as a large collection of polygonal surfaces. Surprisingly, Alias refused to properly render some of these surfaces, instead leaving large amounts of what appears to be noise in the texture image. Even more interestingly, adding or removing other components to the scene (like people or things) could make or break the rendering of the Canoma-provided surfaces, for no apparent reason.

Several hours of experimentation resulted in this: by creating a very flat cube which sits just in front of the non-functional surface, we were able to overcome the difficulty and apply the texture-map to the cube instead. Alias has had no difficulties at all producing flawless texture-mapping every time, from every angle, now that it's looking at a 3D-object instead of Canoma's strange (and unexpected) 2D output.

In short, the interface between Canoma and Alias has failed to impress us. We were expecting that these two software packages would be able to easily interoperate due to the powerful shared data format. Our expectations were far more rosy than the actual performance of the products.

Our overall experience with Alias have been mixed: it is a powerful tool once you figure out how to use it. However, the learning curve was steep and frustrating, at least in the beginning; soon enough we also realized that all of the advertised converters for Alias were either hard to use or (more often) are only partially functional.

Due to our initial decision to use the Blue Moon Rendering Toolkit for our rendering software, we planned to convert all scenes to the RenderMan RIB format before the rendering took place. Having settled on a rendering engine, we then needed to find a modeling tool which we could use to produce all of our 3D models prior to rendering.

Our initial hope was AC3D -- a freeware/shareware modeling tool advertised on the course webpage. We were quite hopeful, because the tool was multi-platform -- thus freeing us from the need to fight for

scarce O2 machines, and (even better!) freeing us from the need to work in the computer cluster to all hours of the night. Its other large benefit was its simplicity -- because we intended to use cartoonlike models and imagery, we weren't anticipating any need for extremely advanced modeling tools.

However, we found to our surprise that AC3D does not exist anywhere on Athena. The software itself is shareware, and registration requires sending a gentleman in Britain the sum of 25 pounds. Without a registered version, the software was crippled to the point of being useless.

Next after AC3D we considered using IV, since it was an environment with which we were already quite familiar. Our expectations in this system rested on the advertised IvToRib conversion utility, which promised to take any .iv scene file and convert it to RenderMan format for our rendering engine's use. However, IvToRib proved to be worse than useless -- it wasn't readily available on Athena after all, and once found, it was unable to even convert something as basic as a cube! Without a working conversion tool, IV had to be abandoned as well.

Lastly, we came to Alias itself. Since there was quite literally nowhere to go from here, we gritted our teeth and prepared to stick it out.

C. Rendering

[Describe the technique we eventually decided to use for rendering]

We wanted to make use of BMRT's networked render farm setup for the massive amount of rendering required. To our surprise, the conversion tools largely do not exist, and those that do generally do not work as advertised. In fact, after tens of hours of hacking, we have not found a complete conversion tool that correctly handles the texture maps we require, to or from Alias. The entire point of using BMRT was simple: it would make rendering fast and efficient, because BMRT will happily run on Solaris, Windows, Linux, and SGI. We were hoping to use a combination of BMRT and AC3D (another modeling tool that runs on multiple platforms) to reduce our dependence on scarce SGI O2 machines, especially near the end of the term.

Ironically, our efforts to make life easier at the end of the road have led to these huge bottlenecks. Alias is, after all, the third modeling tool we have considered, and the second whose conversion utilities we have battled with. The difficulty hinges around the fact that BMRT takes RenderMan .RIB files as input, and Alias does not have native support for the format. The best we can do is save the scene, run the AliasToRenderman converter, and cross our fingers.

Again, we hoped that the conversion utility (AliasToRenderman) would work as advertised. Persuading this tool to run at all was considerably easier than IvToRib, and our initial impressions and simple tests were good -- it was able to handle all of the geometry we tried with aplomb. Our only concern was that it was unable to properly handle texture-mapping of NURBS surfaces. We decided that we could live with this limitation, given that it handled texture-mapping of cubes properly, and we thought that Canoma would output 3D structures like that.

Unfortunately, the limitations of AliasToRenderman were continually bumped, culminating in the generation of invalid .RIB code in response to a scene from Canoma which had been loaded into Alias. This failure effectively destroys any and all hope of using the Blue Moon Rendering Toolkit.

The end result is that circumstances have effectively forced us to use the Alias renderer utility, which is only available on the SGI machines at Athena. The work invested in producing render-farm tools for BMRT (including a dispatcher, PERL servlets, and a simpe network protocol) was completely lost.

D. Image Masking/Dynamic Scene Generation

Unlike the troubles we encountered with conversion tools for the renderer, our planned image diffing and masking worked out quite well. We are pleased to see that one of our design ideas was able to come to fruition without being crippled along the way by half-functioning conversion utilities.

The design problem we were tackling with the image-diff technique was this: given 10 scenes and 10 things which might be in the scene, how can we get high-quality, raytraced images for each possible set of things within the scene? We couldn't raytrace on the fly, because that is unacceptably slow for the user. We couldn't prerender every possibility, because that would be 10 * 210 images -- potentially possible, but not very scalable!

The solution we chose was to render each scene once empty, and then once with each individual object (and only that object) in it. In the case of our example, it would require 110 prerendered images -- for each room, one empty image, and one for each object.

Then the difference between the one image of the empty scene and each of the occupied scenes is computed, and stored away as a transparent image. Later, to show the room occupied by a certain group of objects, the GUI need only load the empty scene's image and then overlay, one at a time, the diffs for the objects which are present.

This technique assumes that no two objects ever overlap each other (including their shadows). We have found, to our surprise and satisfaction, that these criteria are reasonably easy to satisfy.

As an illustrative example, we present the scene 34-101, along with two things which could be in the room -- a sleepy student (asnooze on the desk) and "grouchy," an upset-looking man with a bat. First, here is the rendered scene with both characters present:

Figure: The scene with both characters present

In order to handle all possibilities, we would need to naively prerender four images -- one with nobody in the room, one with only sleepy, one with only grouchy, and one with both. Using the diffing technique, we can get away with three!

First, we render the empty scene and store that:

Figure: The empty scene

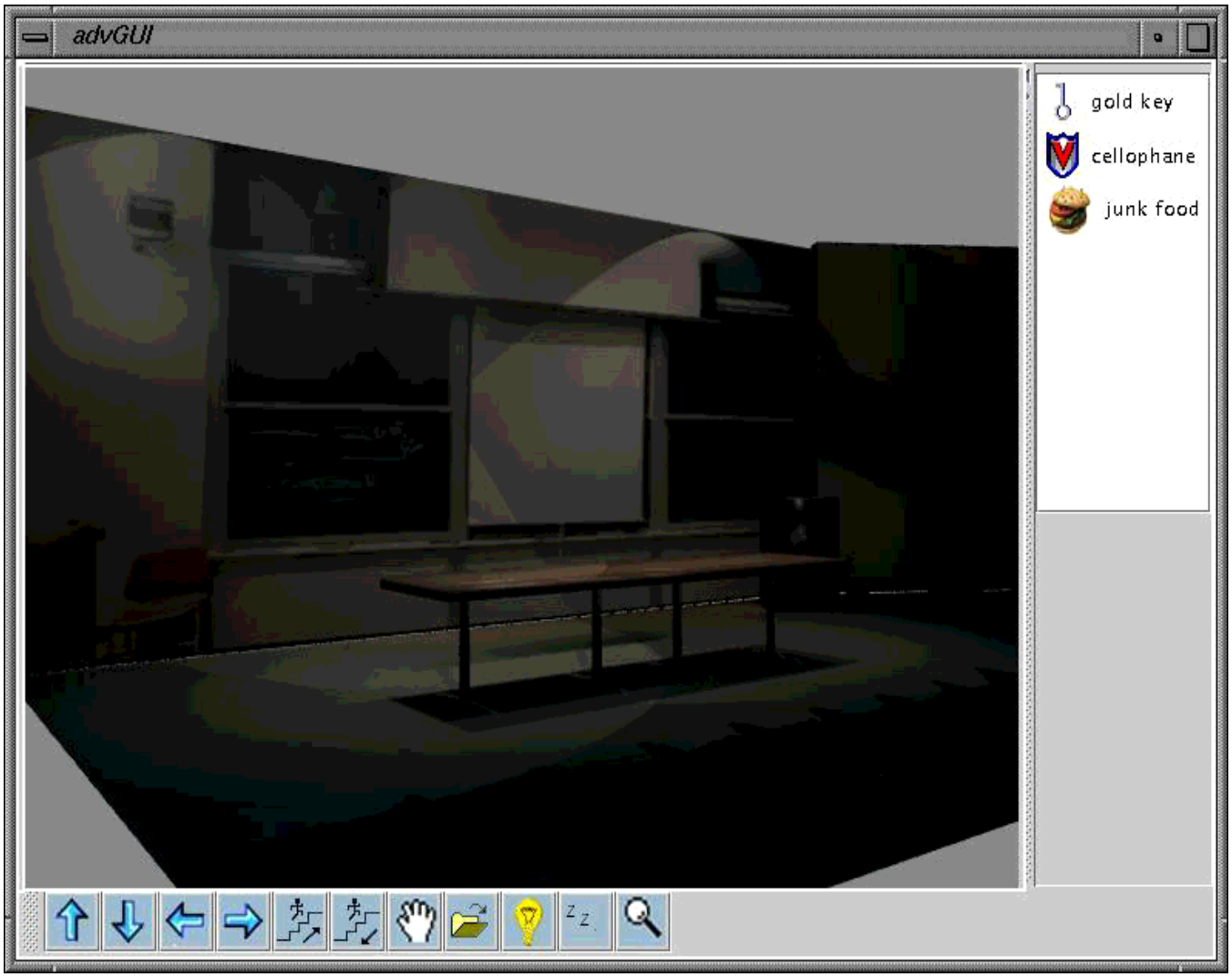
Next, we render the scene once with only sleepy. Using a short utility we wrote called tifdif, we extract the pixels which have changed and make all other pixels transparent. As an intermediate step, we make the "transparent" pixels a hideous green, 0x0BAD2C, and later convert that color to transparency. In order to save space, you will see a clipped version of this mask in our sample images. In practice, we've found that using a small "change threshold" cuts down dramatically on the number of saved pixels and makes for a cleaner mask. Here is the mask for sleepy:

Figure: The image-mask for Sleepy in the given scene (The actual masks are the same size as the image, which ensures that the changed pixels are easy to locate and no offsets are required.)

And here is the mask for grouchy:

Figure: The image-mask for Grouchy in the given scene

With those three images, the GUI is able to reconstruct any of the four possibilities by first loading the empty scene, and then applying zero or more of the image-diffs based on whatever is in the room at the time. Here are two shots of the GUI itself, once with the room occupied, and once with the room empty:



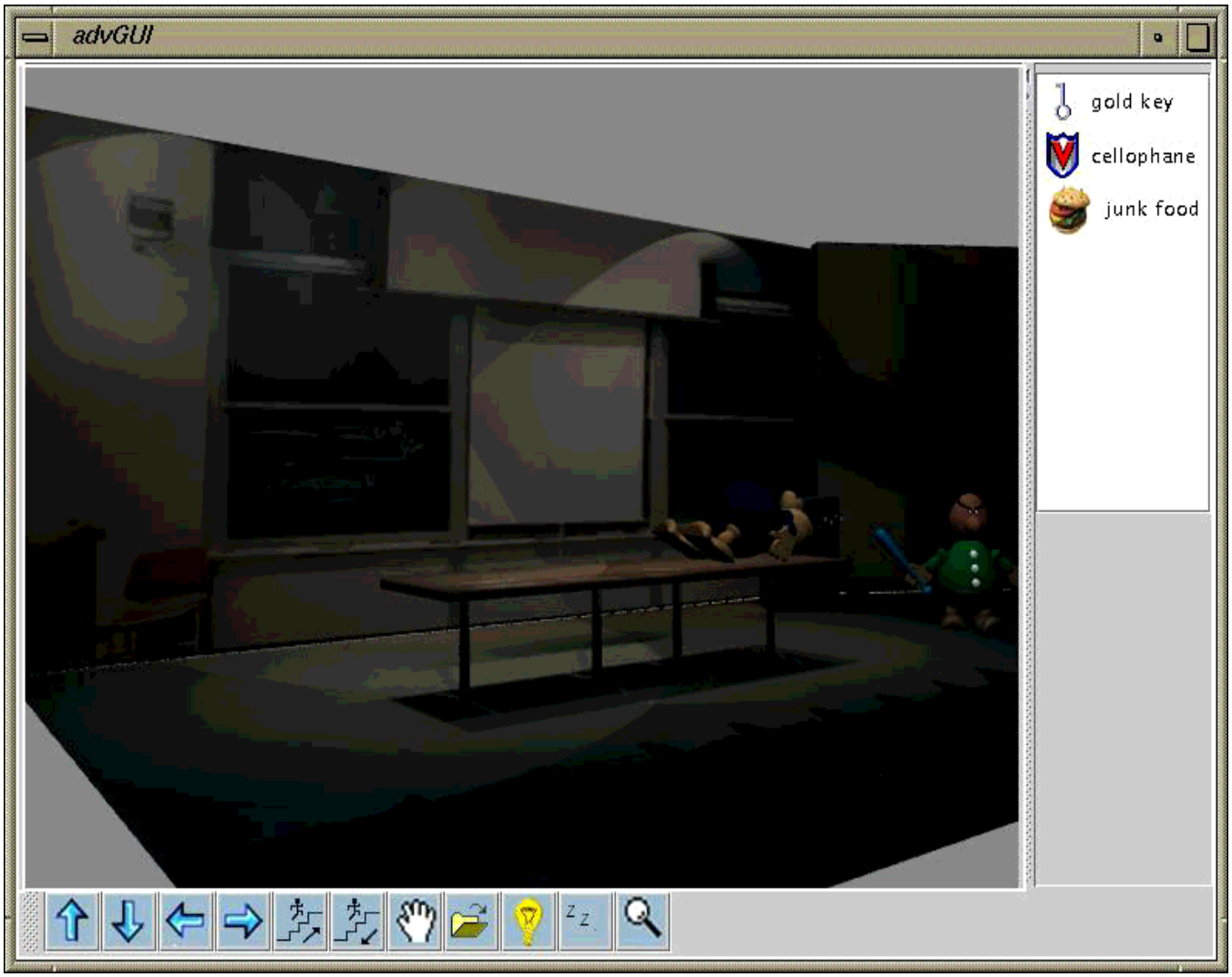


Figure: Screenshots of the game with the room both empty and full

Notice that the image constructed using the diffing technique very closely resembles the result of rendering the image in its entirety.

E. Conclusion

We made satisfactory progress on this project from assessment and planning to tool integration and actual development within the time frame allocated. If given more time, we can almost infinitely extend this project in two aspects: extensions to the adventure game itself and enhancements to our graphical interface. The object oriented design of the adventure game makes it easily extensible by making more scenes, people, objects, and creating new types of behavior and interactions between people in the game. Similarly, we can enhance the graphics by possibly allowing changes of points of view in each scene, animating more scenes, and adding more interesting animated cut scenes. By doing so, we turn the game from its current storybook style into something that contains more action, like a commercial role-playing game. In addition, we can explore better modeling and rendering techniques to enhance the quality of the graphics.

Overall, the engineering challenges we encountered (porting the adventure game, building the render farm, writing the image differ/masking tool) were not as frustrating as the tools related obstacles, such as learning to use Alias Wavefront, making Canoma output compatible with our other tools, and struggling to work around the lack of working converters. When the tools did cooperate, working on this project was actually fun. Most of the non-frustrating work involved designing and modeling our objects and incorporating objects and scenes so they can be masked at runtime. Finally, the most satisfying part was connecting the Java front end and seeing results of our work -- dramatically more interesting than the original text-based adventure game!

Individual Contributions

Kyle Ingols

- Tools assessment; trying to get the converters to work
- Wrote the Render farm
- People modeling (Alias)
- Wrote image diffing/masking tool

Aileen Tang

- Wrote the Java version of adventure game
- People/objects modeling (Alias)
- Scene modeling (Canoma)
- Took digital photos for scenes

Lawrence Wang

- Wrote Java front end
- Objects modeling (Alias)
- Scene modeling (Canoma)
- Integrated game engine and image display with user interface

Group Effort

- Rendering

Lessons Learned

A. Software Engineering Issues

Our primary achievement in terms of software engineering came with the already thoroughly-discussed technique of image diffing. We wanted to come up with some way to get high-quality images without sacrificing the speed of the game. Our first instinct, of course, was to prerender absolutely everything, but that becomes quickly impossible in terms of both prerendering time and storage space. The image-diffing solution we came up with is a good compromise -- by reducing the storage requirements and prerender requirements drastically, we were able to make the game feasible, and at a miniscule runtime performance cost. Other little side-projects, such as the PERL-based distributed rendering

driver, were interesting in their own right but unfortunately didn't make it into the final project due to unforeseen difficulties with support utilities.

B. Project Management Issues

We learned the hard way how one should not over-estimate the usability of available tools and resources. All of the converters should be sanity checked early on to make sure they satisfied our project's requirements before any substantial work was done that required dependence upon any particular converter. This involves more project planning based on assessment of all of the pieces of the project early on, so we do not run into obstacles late in the game.

For example, we wrote the render farm before we had scenes and objects to test whether the Alias to Renderman converter really worked. As a result, the effort of writing the render farm was wasted because we made the incorrect assumption that the tools would behave as advertised. We needed a more structured schedule that coordinated the specific order in which tasks should be completed instead of having things happen concurrently.

We also learned about finding new tools to do the job (e.g. Canoma) and to work around tools that did not work as expected (e.g. resorting to a different rendering scheme when AliasToRenderman could not properly handle Canoma outputs). Finally, the standard lesson involving project management was correctly pacing and scoping a month-long project like this one.

Acknowledgements

We based our Java game engine on the Scheme object oriented Adventure Game system used in the 6.001 problem set. Charles Lee helped us come up with the image diffing and masking approach, along with help in tool assessment (using Alias Wavefront, Canoma). We received help from Damian Isla when we got stuck on Alias problems. Kari Anne Kjolaas provided the tutorial material on Canoma.

Bibliography

- Java in a Nutshell by David Flanagan, 2nd Edition, O'Reilly & Associates.
- 6.001 Problem Set 6, Fall term 1999.
  URL: http://sicp.ai.mit.edu/psets/ps6web/ps6-oop/ps6-oop.html.
  For the adventure game code, see Appendix.
- Countless miscellaneous webpages.
- The freeware *libtiff* library, used to make the tifdif utility.
- Java documentation:
  http://java.sun.com/products/jdk/1.2/docs/api/index.html
  http://java.sun.com/docs/books/tutorial/2d/index.html
  http://www.gamelan.com
- The ever-helpful 6.837 Teaching Assistants.

Appendix

Original Scheme source for the adventure game:

- objsys.scm the object oriented system
- objtypes.scm definitions of the objects in the game
- setup.scm setting up and running the game: creation of the game world and object creating and installation into this world