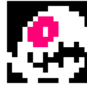
Bubble Trouble Bobble 3D Challenge 3000

6.837 Final Project Report

baron-von-blubba@mit.edu



Roger Hanna
Ryan Williams
Chris Elledge
Paul Elliott

December 6, 2002

Abstract

"Bubble Bobble" is a 2D arcade game by Taito. We made a new version of the game that uses 3D computer graphics techniques to visualize the gameplay. We wanted our implementation of the game to have the features of the original game plus several graphical enhancements and to allow future extensions of the game into a full three dimensional space. We achieved a large number of our goals and successfully created a playable version of the game. Through designing this game, we learned to work with OpenGL and how to make various parts of a game interact to build a playable system.

Introduction

Many in our generation have played the game Bubble Bobble. It was a fun arcade game that was so successful that it was ported to almost every platform that exists. It featured little dinosaurs blowing bubbles to complete puzzle-like levels. Roger happens to be a huge fan of the game, and thus when the subject of 6.837 final project came up, we came upon the idea of doing a remake. The games numerous enemies, power-ups, bonus items, and levels combined with a cartoonish cooperative gameplay make it have great replay value.

We fit into the category of "people who like computer games" and it was no great surprise to discover that we're also interested in creating them. For everyone except Paul, this was our first time making a game of this scale.

Games are in some ways the driving force behind computer graphics. They sell better if they look amazing, so games developers are always pushing the envelope of realtime graphic effects. There's also a drive for more realism, so more detailed physics engines are made. Though it's rather commoditized nowadays, the games industry is very interesting to all of us.

Even though we barely scratched the surface of gaming, we feel that we're no longer beginners in the art of game-making. We've created a game that anyone can play and have fun with. It's got several levels, several enemies, a scoring system, and plenty of interesting twists. Our game features all of the best elements of the original, while adding 3D freedoms and graphics.

Goals and Achievements

We enumerated several goals in our proposal, and would like to revisit them to investigate our success in accomplishing them.

Refractive, transparent bubbles

Goal: The bubbles need to be able to capture enemies and pop believably. They need to be influenced by wind currents. In addition to being translucent, they should also refract light, and be glossy.

Achievement: We accomplished all of the physical elements of the bubbles - i.e., they are blown about the level, they capture enemies and pop. In developing the game, however, we noticed that, among other things, we had a black background, which made rendering refraction unnecessary. The bubbles served their purpose admirably without the reflections, and indeed that might have jarred with the flat-shaded style of the game. They are translucent, so that enemies can be seen when captured, and that is sufficient for us.

Cel Shading

Goal: Cel shading is a style of rendering that uses cool-to-warm shading with silhouettes for player characters, bubbles, enemies – everything that's interactive in the game. This gives it a comic book or old video game feel, while not detrimenting from the quality of graphics.

Achievement: One of our first design decisions was to put cel shading on the very back burner. It is very little more than eye candy, and the essential elements of the game can be completed well before. We did not get to investigate cel shading in our game, but deliver a more finished product as a result.

Wind

Goal: Wind fields are statically distributed throughout a level, and they contribute heavily to the gameplay by blowing bubbles around. They are constant-momentum regions. They should somehow be visible to the players through the motion of indigenous dust particles. They should push movable objects around at a speed proportional to the mass of the object. This makes new levels more usable, because it gives the game players more information.

Achievement: The static wind fields do indeed behave exactly as described. Blowing a cloud of bubbles results in a moving flow of bubbles through the wind currents. The levels are designed in such a way that bubbles are blown somewhat purposefully through the level, so learning "which way the wind blows" is very helpful in successfully finishing a level. Particles were another item we put on the end of our feature list. The bubbles act as particles enough to indicate wind direction.

Viewpoints

Goal: In order to make the game playable in its original side-on form, the game should have the option to be played with a fixed viewpoint. But to realistically play in a 3D environment, you need to resolve occlusions of the characters, so other camera angles will need to be implemented.

Achievement: The game is available in fixed-viewpoint form, by default. It is possible to enable mouse control of the viewpoint. In user control of the viewpoint, the camera points resolutely at the center of the level, and the user can pan and zoom around the level.

The world

Goal: Bubble Bobble consists of a series of levels. Each level is toroidal, meaning that moving off one side of the screen means moving in to the other side of the screen. In practice this only applies to the top and bottom of the level. Each world consists of a set of blocks, some with persistent winds, some with structural elements, and most empty. In our 3D translation, they would be cubes.

Achievement:
Each level is indeed toroidal, and every face of the bounding box of the level wraps to a corresponding opposite face. Our worlds are built a bit differently than we had originally anticipated. They are described in a series of plaintext files that represent the winds, items, and geometry, and another file containing a colored mesh to display (not all maps have geometry that corresponds with the visual appearance of the level).

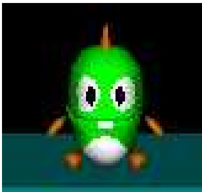
A game

Goal: We would like to make a game that satisfies fans of the original Bubble Bobble while remaining extensible enough for us to continue to tinker with on our own time in the future.

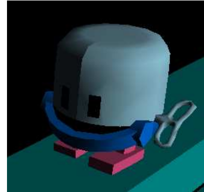
Achievement:
We have created a game that has the physics, scoring, and general appearance of Bubble Bobble (though one observer's comment was "It's just like Bubble Bobble, but with uglier guys!"). It looks reasonably professional and is relatively bug-free. We have also left plenty of room to upgrade ourselves - for example, the entire physics model could be replaced with a more "realistic" one without affecting the rest of the program.

2.1 The game in brief

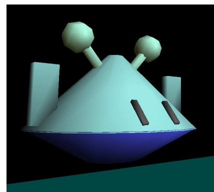
The characters that we have designed are:



Dino



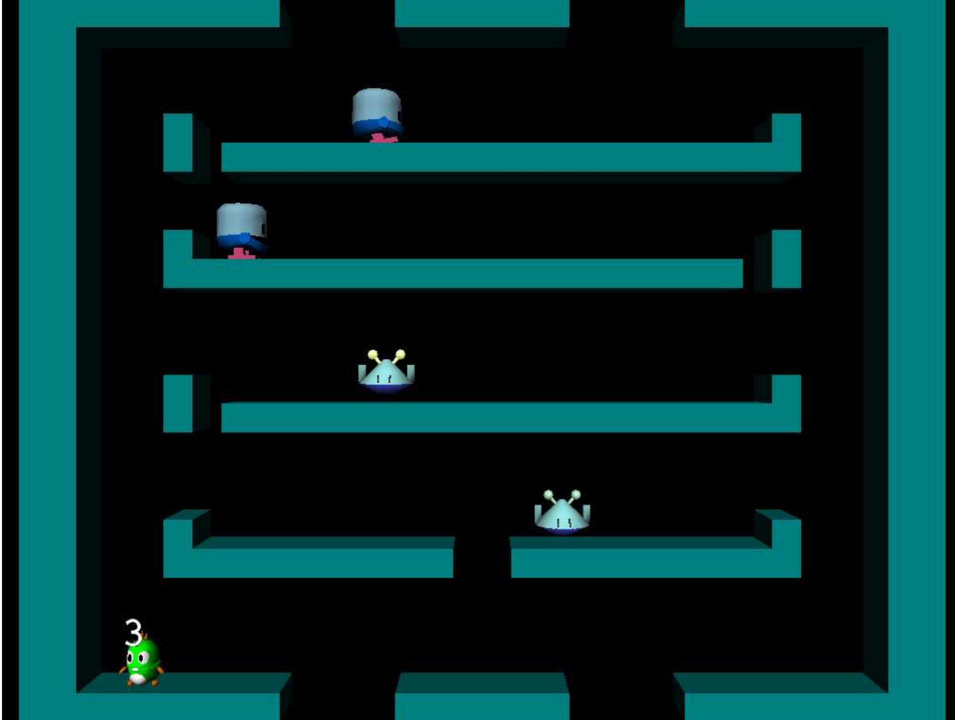
Windup



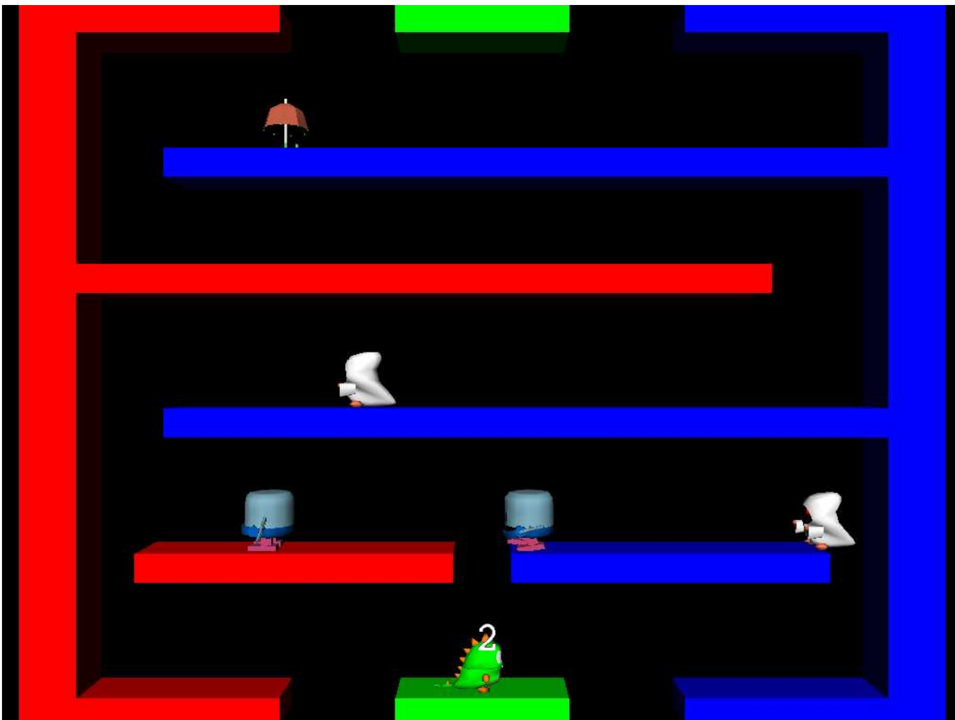
Alien

Whitemage

Our first level looks like this:

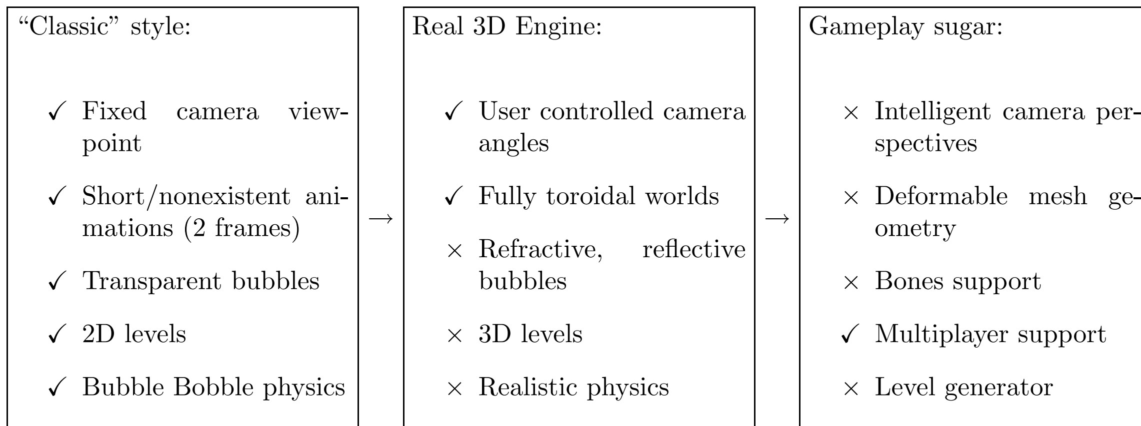


This is our second level:



2.2 Phases of construction

We viewed this game as going through several “phases” of construction. The earlier phases represent reduced functionality – as we progress with the game, we intend to add features and power. We expected that just getting the first parts would be very hard, and we were right. The parts that we have completed are highlighted below.



As is clear, we made our way about halfway through the phases. We are perhaps close on many of the other goals. For example, it would take only a little work to add support for levels that are deep enough to walk in and out relative to the camera plane (we would only need to add controls to do so, create levels that take advantage of this, and add a more intelligent camera view). Realistic physics, on the other hand, are very far away from completion.

2.3 Components

The overall design of the system has changed only in subtle ways since its inception.

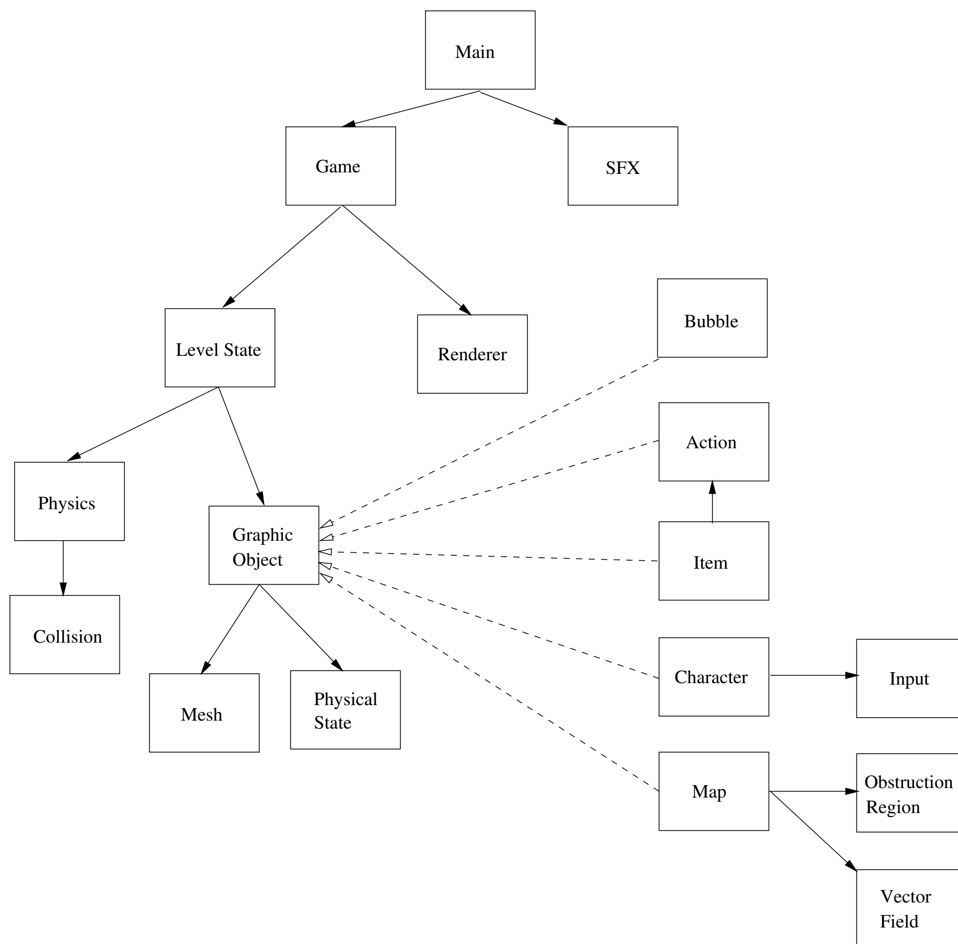


Figure 2.1: Object model - Dotted lines are “is-a” relationships.

GraphicObject:
An abstraction for a "thing" in the game. Keeps track of how to render the object given a renderer, and maintains a physical state.
Methods: draw(renderer), update(Δt)

Dynamic Object:
An abstraction for objects that move and act. Maintains a physics state that includes velocity, position, and mass, as well as a state machine for making decisions.

Map: Contains the geometry of an individual level. Allows for some way of addressing individual maps, for changing levels in the course of the game. Loads the obstructed regions and the vector field.

Bubble: Keeps track of the special properties of bubbles - they are not rendered like normal entities. Handles pseudo-environment mapping of the bubbles, animations.

Renderer:
Renders internal data representations of media into operating system calls. This renders meshes to the OpenGL subsystem, sprites to the screen, and audio to the sound system.
Methods: renderMesh(mesh), renderSprite(sprite), renderText(string)

Input: Gives commands to the characters for movement. Handles the player's input, and the AI controls of the enemy characters.

Character:
Maintain the state of each character. Control animation based on state. Convert input into physical movement.

Items: Hold information about the special property of each item, such as points and special actions.

Game State:
Stores information that's persistent between levels, such as the score of each player, the player models, input mappings, and in-game options. Handles transitions between levels. Detects when the game is completed.

Level State:
Keeps track of all the objects in a level, initiates level-wide animations on behalf of special items. Sets up physics for each level. Detects when the level is completed.

Physics: Detects collisions between graphical objects, and with the level. Performs actions based on the types of object colliding. For example, an enemy colliding with the player will result in the player dying, and a active bubble colliding with an enemy will result in the enemy being engulfed. This also performs the Euler approximation of the equations of motion. The game is not especially concerned with adherence to "real-life" physics, so very crude approximations that are consistently graphically reflected are sufficient.

Main: Assembles all the other modules together so that they can run. Controls user options.

Individual Contributions

Chris Elledge

Out of Game:
: Sets up in-game components, configures game libraries, handles user configuration

Item Game Objects:
: handles item states, special properties of items, actions generated by items.

Action Game Objects:
: allows other objects in the game to affect the overall Game State.

Audio:
: plays a background music track and provides audio objects to allow other game objects to play sound effects.

Accomplishments:
: The Out of Game system controls the setup of all the components that are needed to begin a new game and controls the thread of execution during play. It achieves some platform independence for video, audio, and input by using SDL (the Simple DirectMedia Layer). Several other extension libraries for SDL were also used to build an audio mixer for the game that could also decode and play MP3 music files.

SDL is a library which normally requires installation of it and any of its extension libraries on the system which the game will run. Since our desired development and testing platforms were the athena-linux machines, it was impossible to perform these installations. We also would prefer not to make the user install any extra libraries on their system to play our game. I forced the libraries to build into a directory under my control and packaged them so that they could be placed into the CVS repository to allow us to link to them at compile time and to run against them at runtime. This technique also allows us to distribute a version of the program that doesn't need the libraries installed, but this will only work on an Intel system running the Linux kernel 2.4. I also made a target for our makefile that will build a tarball for our executable with the needed libraries and data files to distribute a working binary.

The Out of Game system regains control of the thread of execution at the end of every frame of game updates and rendering. I have the system analyze the SDL Events that have occurred since the last frame and update the Game Object accordingly. These events include key presses for player controls, mouse camera controls, and toggling of sound, music, and various other parameters. The elapsed time is estimated and the game is updated with the elapsed time and the renderer is asked to render the scene based on its frame skipping rules. Finally the thread sleeps for the remainder of the time slice so that the audio thread can handle mixing and playback.

Items are a subclass of Dynamic Object which contain Actions that are to be performed when the item is picked up. These Actions may do any number of things depending upon the item, but all items have a Points Action which gives the player points for picking it up. I concentrated my effort in Items on allowing any arbitrary combination of 3D meshes, actions, and point values to be given to a particular item. A single database file contains a list of the above information for each item, and the item is constructed from its definition in the list. This allows for easy changing and addition of items, models, and actions for the large number of items that a complete version of this game will have.

Actions are a design decision that we chose to use to allow Graphical Objects in the game to affect the overall game state. It centralizes the knowledge about the Game Object maintaining the game to a single class and simplifies interaction. There are several Action types that include popping, engulfing, bubble generation, level skipping, power ups, and ones that kill all the bad guys. Each of these extend the base Action interface. The ones that can be added to items also have a factory that generates these actions based on their type as specified in the Item database.

Audio for the game was mixed in its own thread through the use of SDL Mixer. There are multiple channels for sound effects and a channel for background music.

The music is played as an MP3 using the SMPEG library piped into the mixer. I made audio objects which hold loaded WAV sound effects. A call to the object to play grabs a free channel and performs the setup necessary for the mixer to play the sound.

Paul Elliott

Renderer & Mesh Object:
: Displays an in-game object's meshes in a 3D environment with our desired look.

Static Objects:
: Maps are unchanging objects which need to setup the bounds of play, specify hazards, and points of special action. They also contain wind information.

Accomplishments:
: The renderer is implemented using OpenGL along with a support library called TexFont. TexFont uses a font texture from a txf file to render text using textured quads. I generate a set of transforms in the text rendering process to draw the text at one unit high (not the default) as well as drawing it in the X-Z plane instead of the X-Y plane.

I also support transparency in the game through alpha blending. This has the limitation that transparent objects need to be drawn last. Since only bubbles are transparent, this isn't particularly constraining.

The renderer also supports Ambient, Diffuse, and Emissive lighting. Specular support is almost there but most models currently have a specular component by default, which is incorrect, so specular light is disabled right now. My version of Rhinoceros always set the specular color to match the diffuse color, which creates undesirable artifacts.

The renderer uses display lists on a per VertexSet object basis. The VertexSet corresponds to a single object in a mesh. For instance, in the Dino mesh, there are 21 vertex sets corresponding to it's eyes (4), feet (2), hands (2), belly, body, head, tooth, tail (2), and spines (7). Each one is is turned into it's own vertex set, so each character can be animated based on it's parts.

Currently the renderer only uses one infinite light source, but we have the ability to do something with point light sources for fire, whenever we support fire.

The renderer sets the camera location per frame, allowing for dynamic repositioning of the camera by the user. At the moment the user can only change where the camera is located, but not where the camera looks at, so the camera will always be looking at the level.

The renderer also supports setting the mesh's material dynamically. Currently this is supported by switching back to immediate mode rendering, though in principle display lists could be used if the architecture was slightly modified. This feature is used to make the antenna of the Alien glow as well as to make the bubbles turn red and pulsate when they're about to pop.

Meshes are loaded from 3ds (3D Studio) files. These files contain geometry and material information. Normals are computed as the average face normal across a vertex set for each indexed vertex. The mesh loader is based on a c3ds loader library and has been augmented to be a bit more flexible as well as supporting several more constructs, namely materials and some additional geometry information.

Meshes also implement procedural animation. The animation procedure is selected based on the file name of the loaded mesh. The procedure, then, knows about all the objects in the mesh it's written to animate.

Animation has both an ambient time counter, which can independently follows the elapsed time, as well as an animation counter. This lets ambient actions such as the crank spinning on the WindUp to always be continuous even when changing the animation rapidly. Each animation has a animation length associated with it which rolls the animation time around, allowing easy application of periodic animations. It also has a flag associated with whether the current animation time has been rolled

around yet or not. This is used for jumping to transition from jumping to flying automatically.

There are currently 4 animations, one of which was provided by Ryan: Animations coordinates are taken from the native mesh and the mapping between each animation part is manually mapped to the object UID in the mesh file.

Dino: (6 animations)
Idle animation - idle animation wags his tail and if you sit long enough he will wave his arms.
Dying - all characters die by spinning about their axis. The dino also crosses his eyes by drawing his normal eye twice at +-30 degrees.
Walking - The dino struts by moving his legs in an alternating fashion. His tail matches the period of his walking (instead of his ambient walking rate) and his body sways from side to side as he walks.
Jumping - The dino picks up his feet and slams the ground. his tail falls down from the force of flying up and he also looks up.
Attacking - The dino opens his mouth for a moment to let out a bubble and then closes it again.
Falling - When falling, the dino's tail raises up from the lack of forces pulling it down and it also looks down.

WindUp:
(4 animations)
Idle Animation - The WindUp's crank is always running, as is it's mouth.
Dying - The WindUp dies by spinning
Walking - The WindUp has a mechanical walk.
Jumping - The Windup jumps by lowering down on it's legs and then springing off the ground

Alien: (2 animations)
Idle Animation - The Alien's antenna glow
Dying - The Alien dies by spinning

Frame time estimate - uses a Kalman filter to estimate the average time between frames, which is used to compute the frame rate. It's also used to decide when frames should be dropped.

I also modeled a number of meshes: The Dino, Umbrella, WindUp, White Mage, Level5 & Level6

Roger Hanna

Game & Level State:
Handles the state of gameplay within a level and persistent state of gameplay between levels.

AI & User Character Control:
Accepts inputs from user or AI to control the actions of characters, enemies and the camera. Also runs a state based AI for enemies.

Accomplishments:
Taking the game state and characters gave me control of what the user sees and hears, choosing where to draw what, what to draw, selection of animations, timing of character control, item appearance and AI.

For the game, I generate the animations for the inter-level transitions, as well as choose which items appear in a given level, deterministically based upon the previous actions of the users.

With the characters, I generalized all the characters into one class, which selects which animations they should be drawing, as well as takes a given set of keys up, down,

left, right, attack, jump and chooses the movement for the character, independent of who the controller is.

While unnecessary for now, this abstraction makes it better if there were ever more than 2 teams, such as a player vs player game, rather than cooperative mode.

Also, there is a cinematic mode for the characters, where no life information is draw, but an additional mesh is drawn around the character and they face the camera. Currently it is only used for the inter-level transitions, but it could be expanded to add "plot"

I chose to use a fast protocol for the AI, where you have access to the PhysicalState of yourself and one target. With only this information, I have generated the original 3 AIs that existed. One for the alien, which hovers around on the ground. One for the remaining ground characters, which runs around and jumps. One for flying units which bounce around the stage.

However this format of AI is still much broader than just this. You could, for example, create an AI which creates a tractor-beam force upon its target, or other just as interesting controls.

For now, the users control over their Dino is fairly limited, only having one more key than the original game; however, the game records durations of keystrokes and can potentially make for a more responsive keyset. Making the game have better play control and become more enjoyable for its players.

Other than writing code, I also got to experience 3D Studio Max and generated 2 models, an Alien and Level6837. I located the remixes of Bubble Bobble (they are all from http://www.mp3.org if you search for Bubble Bobble) and gathered the sound effects. Since we still don't have a generator for levels, I created the vector fields and obstruction regions and level information for the levels that currently exist.

Ryan Williams

Physics: Handles movement of level objects and application of forces. Detects collisions and resolves their interactions.

Graphic, Bubble, and Particle Objects:
Handles the interaction between game objects and the render. Also handles special properties of Bubbles and Particles in game.

Accomplishments:
Physics was much harder to implement than I'd thought. Perhaps this was because I had no knowledge of how to do it beforehand. I was convinced that a realistic system would produce the most satisfying gameplay, and I was mostly right. However, to accomplish a lot of the physics in the game, nonrealistic assumptions were used (such as bubbles not colliding with the world). Right now some things are "realistic" (such as gravity and the viscous drag coefficient), while others are not (jumping through ceilings). Not only is each character checked for collisions with the world, but the face that the character hits the world with is recorded in a flag in its state. Velocities are unlimited, though in practice they are cut down by the viscous coefficient.

When things bounce off of each other, they aren't actually separated, they merely apply a momentum to each other proportional to the inverse of the distance between them. This is why when a bubble is blown against the player, the player gets knocked back very slowly by the bubble while the bubble is essentially stopped. Originally this was a system to place their boundaries outside of each other and apply a "realistic" momentum transfer. However, that looked wrong for the springy bubbles.

A lot of my development was plagued by a buggy library. I used freesolid, an open-source library for generic collision detection. It was my hope that the actual geometries of characters could eventually be used for collision detection, but it was not to be so. The library had no end of errors. About one-third of the time, it would simply fail to detect a collision - as a result, bubbles often stuck together, then were blown violently apart when the collision resolution displaced them. Also, since it only had a C API, every single translation of any item had to be carried out through Physics,

causing no end of hassles. In the end, to increase reliability, I threw out freesolid and used instead a very simple $O(n^2)$ algorithm that checks the distance between the centers of objects (a bounding sphere approach). For the very regular and few items in our game, this performs quickly enough. And since only spherical objects (bubbles) have any sort of meaningful collision, the sphere approach is very apt.

There is an incredibly hairy conditional structure designed to weed out the various types of collisions. For example, a character colliding with a bubble can either pop the bubble, jump off of it, or push it around. If the character is on top of the bubble, he should be able to jump off of it if the jump key is being held down. However, if he is not attempting to jump, the bubble should pop when landed on. Touching a bubble with the back of a character should pop it, whereas touching with the belly should push it – unless, of course, they are moving fast enough to pop the bubble. The bubbles are still a little to easy to pop relative to the original game, but that will only require tweaking to fix.

Collisions with the map are resolved via the bounding boxes of objects. The level itself is two units deep, as are most of the characters (the exception being the final boss). Thus, it becomes possible to merely check the corners of the bounding boxes for collisions. In addition to checking the corners, the middle of each edge and face is checked. Each check is a constant-time lookup in the Obstruction Region object. During these checks, the physics engine determines which if any faces are colliding with the level, aligns those faces more precisely with the level (because when such a collision is detected, the object is usually pretty deep into the level's geometry), and nullifies the object's velocity in the direction of the colliding face (this is necessary so momentum doesn't accumulate).

Creating the GraphicObject and DynamicObject classes was something of an exercise in trial-and-error. I knew what they needed at minimum because of our design, but as to what else was needed, I could only guess. My primary enhancement was the development of a state machine usable by the various classes; I got the idea from Paul. It's little more than a variable holding a function pointer, that gets called every update. Thus, when the object changes states, it changes the function pointer. Perhaps it would have been more full-featured if it had included a table of state changes, but as it is, it's used to "animate" the creation of bubbles and a few other things. Most of my effort went into a little carry-on class, PhysicalState, that goes along with every GraphicObject. PhysicalState keeps track of every variable related to Physics calculations, and in particular it manages accumulation of forces. I initially thought that applying forces for a specified duration would be an important feature, but as it has turned out, we haven't used it at all. Most forces are either instantaneous, or amortized over many timesteps.

Additionally, I made the presentation slides and coordinated the creation of both documents - proposal and report.

Lessons Learned

The experience of making a game gave us valuable experience with the world of real-time calculation. Fortunately, we didn't have to do too many optimizations on the graphics side – those are known to be something of a black art and would have been a huge timesink. We did have to be careful all the time to not choose slow algorithms or allocate huge blocks of memory. Due to the time-stepped nature of the game, every action had to be performed in the context of being executed alongside a hundred other similar actions in the same timestep.

CVS was a key element of our strategy. Chris's Athena machine not-the-hub.mit.edu hosted our repositories since before the project started. It was essential to have a centralized version of the program for us all to work on. Most of us had previous experience with CVS for other projects.

We also learned that it is important to define non-changing interfaces right from the start. After we formed our team, we held several long meetings where we did nothing but draw up interfaces and object structures (much like Figure 2.1). Then we wrote out C^{++} header files for each object, and stuck with them. Of course, as features were added, we had to add more methods, more objects, and more variables. The core calls (such as Physics::updateForces or Renderer::renderMesh) remained essentially unchanged. At one point, we modified the GraphicObject's update method so that it returned a boolean value (to check whether object should be removed from play). Making this very small change, even though we were only two weeks into programming, took about half an hour of searching for references to the method, and we tried to refrain from making core interface changes subsequently.

It was nice to have someone around who was experienced in C^{++}. Many of us had only had previous programming experience with Java or the like, and thus were confused about the difference between references and pointers, how to link libraries, and so on. Paul was able to help us out every time with his immense practical experience. He is also a debugging titan.

We believe that this project went so well in part because we chose such good divisions between the labor. In hindsight it is even more clear. Th four elements of Renderer, Physics, Game, and Out-of-Game were almost completely independent of each other's interface, and so each of us was able to work on his part without the others. Very little effort was spent on synchronizing our efforts. During the last week of the project, we drew up a list of minimum tasks for each person.

At many times during the development, we had to make choices that compromised between perfect extensibility and a working game. For example, we elected to use a integer-aligned, cubes-only method of representing obstructed regions of the level because it makes for constant-time collision detection for objects and is easy to write level files for. A more general implementation would have handled arbitrary level geometry. The procedural animations are hard-coded, and run based on the filename of the mesh. We had hoped to incorporate some form of loading animations, perhaps from the 3DS file format, but that did not pan out and the procedural animations were a good-enough-looking compromise. The most visible compromise was the decision to not add reflectance or refractance to bubbles. We made this call because we felt that adding these features would have provided low impact relative to the effort required. Without these compromises, the game as a whole would not have made it to this polished stage.

Description of Deliverables

We have made available a binary distribution, consisting of precompiled game executables and libraries, and a source distribution, our development environment.

5.4 Build instructions

Binary distribution:

To run the game from a distribution tarball, unpack the tarball with:

```
tar -xzf BubbleTrouble.tar.gz
```

This will create a directory BubbleTrouble. Run the game:

```
cd BubbleTrouble
./BubbleTrouble
```

Source distribution:

Unpack the tarball with:

```
tar -xzf BubbleTrouble-src.tar.gz
```

This will create a directory BubbleTrouble. Compile and run the game:

```
cd BubbleTrouble
make
./Main
```

5.5 Command-Line Options

`-h`	Lists available command-line options.
`-s`	Turns on sound subsystem.
`-f`	Renders in fullscreen mode.
`-d <height>x<width>`	Uses the given resolution for the window (or screen).

5.6 Controls

	Player 1	Player 2
Left	←	D
Right	→	G
Jump	Right Shift	Left Shift
Attack	Right Ctrl	Left Ctrl
Drop	↓	F

`meta+s`	Toggles sound effects, if audio was enabled at execution.
`meta+c`	Toggles mouse operated camera control.
`meta+m`	Toggles music, if audio was enabled at execution.
`meta+1`	Creates a new game with 1 players.
`meta+2`	Creates a new game with 2 players.
`mouse1`	Zooms camera in, if mouse camera control is enabled.
`mouse2`	Zooms camera out, if mouse camera control is enabled.

Acknowledgements

For this program, we used a number of libraries and resources. Without these, this would have been a much more monumental task.

OpenGL - *3D rendering*

SDL - *Sound, image, and user interaction support.*

C3ds - *For loading 3DS meshes*

TexFont - *OpenGL textured fonts.*

3D Studio Max - *Mesh generation*

Rhinoceros - *Mesh generation*

emacs - *Code generation*

The 6.837 Staff - *For supporting us in this endeavor*

File Formats

A.1 Music

Music is loaded from files formatted in the MP3 file format.

A.2 Sounds

Sounds are loaded from files formatted in the WAV file format.

A.3 Items

*.db - Item File Format

items.db describes the all the items used in the game.

Format:

*.db=$ITEMS$
$ITEMS$=$ITEM$ |$ITEM$ $ITEMS$
$ITEM$=$NAME$ $FILENAME$ $POINTS$ $METHOD$
$NAME$=*$char$
$FILENAME$=*$char$.3ds
$POINTS$=int
$METHOD$=int

A.4 Models

Meshes for all GraphicObjects are loaded from files formatted in the 3DS file format.

A.5 Level

A level consists of a Model, and 3 ASCII File definitions: lvl, vf, and or

*.lvl - Level File Format

lvl describes the start locations for DynamicObjects, as well as the other files to use for a particular level.

Format:

*.lvl=$MESH$ VF OR $NEXT$ $ITEMS$ $BUBBLES$ $ENEMIES$
$MESH$=UserMesh "$FILENAME$"
VF=UseVectorField "$FILENAME$"
OR=UseObstructionRegion "$FILENAME$"
$NEXT$=NextLevel "$FILENAME$"
$FILENAME$=*$char$

$ITEMS$=$MAINITEM$ $SECONDITEM$
$MAINITEM$=MainItemLocation int int int
$SECONDITEM$=SecondaryItemLocation int int int

$BUBBLES$= ϵ |UseBubbles $FIRE$ $LIGHTNING$ $WATER$
$FIRE$= ϵ |Fire
$LIGHTNING$= ϵ |Lightning
$WATER$= ϵ |Water

ENEMIES= *ENEMY* |*ENEMY ENEMIES*
ENEMY = Enemy *TYPE int int int*
TYPE = WindUp |WhiteMage |Alien |Whale

*.vf - Vector Field File Format

vf describes the vector field which drives bubbles around the level.

Format:

*.vf=*VF-DIMS VF-DEFS VF-REGION*
VF-DIMS=VectorField *XDIM YDIM ZDIM*

VF-DEFS=*VF-DEF* |*VF-DEF VF-DEFS*
VF-DEF=#define *VF-VAR* {*int*,*int*,*int*}

VF-REGION=*VF-SLICE* |*VF-SLICE VF-REGION*
VF-SLICE= *VF-VAR* |*VF-VAR VF-SLICE*
VF-VAR=*char*

XDIM=*int*
YDIM=*int*
ZDIM=*int*

*.or - Obstruction Region File Format

or describes the obstruction region for a StaticObject.

Format:

*.or=*OR-DIMS OR-REGION*
OR-DIMS=ObstructionRegion *XDIM YDIM ZDIM* '*OR-VAR*' '*OR-VAR*'

OR-REGION=*OR-SLICE* |*OR-SLICE OR-REGION*
OR-SLICE=*OR-VAR* |*OR-VAR OR-SLICE*
OR-VAR=*char*