Luminance Design

Kyle Fisher, Stephen Damm, Greg Logan
kaf020, sad503, gdl420

March 2nd, 2009

Introduction

Luminance is an easy to play puzzle game that requires a new twist on the connect-the-dots approach. This concept leads to a very familiar setting in which to play the game and thus provides a very easy to approach experience. The game is meant to appear to be simple but actual solutions may not be so obvious, and therefore the target market for Luminance is teenagers and up. Different levels of difficulty and the challenge of finding the most optimal way of solving a puzzle results in an incredibly addictive format. This addiction is fed by the random level generator, ensuring that there are always new possibilities and challenges to solve. We believe that this combination will result in high sales and very positive word of mouth.

The team behind Luminance is Kyle Fisher, Stephen Damm, and Greg Logan. Stephen has previous game building experience and is the primary person responsible for the game engine and artistic influence. Greg and Kyle are both experienced with artifical intelligence and will be primarily responsible for the random level generation and level difficulty adjustments. Greg has the most formal knowledge in projects and as such also helps in the development of various class APIs used.

Game Description

The concept of Luminance is to use a level-specific set of tools to direct a beam of light through a set of obstacles in order to illuminate one or more goals. Each level has at most one emitter which acts as the starting point of the light beam. Light will always travel in a striaght line while unobstructed. There are various tools that will be available for the player to use in order to change the light beam in certain ways. Mirrors will reflect light at 45 or 90 degree angles and prisms can split the beam in order for it to hit multiple targets, for instance. Other ideas include: Certain goals requiring a specific frequency of light thus forcing the player to first direct the beam through a filter, the mirrors themselves acting as filters, and starting with multiple frequencies of light and combining them with a lens to get white again. The latter of these ideas would require the stage to have some unmovable tools incorporated as part of the level design or generation.

There will be two modes to Luminance. The first is a set of 5 to 10 levels designed to train the

player and introduce them to all the concepts they can expect to encounter. The number of these levels, therefore, depends on the number of concepts. The second will consist of randomly generated levels. These levels are expected to vary in difficulty and will provide a continuous stream of entertainment once the initial set of levels is complete. A goal for the generated levels is to both obstruct trivial solutions to the level and to not include so many obstructions that the solution will become impossible.

Luminance holds a lot of potential in respect to sales and reputation. The game will appeal to a broad audience and have enough lasting appeal to keep people's attention and their word of mouth going. Development costs are relatively low; all the tools needed to build the game (Visual Studio 2008, XNA 3.0) are available for free. The game uses very simple assets and can use only the basic forms of collision detection and other physics. A bottleneck in development will be the level generator, but with 66% of the team having a strong background with artifical intelligence, this bottleneck shall be overcome.

Art Design

As alluded to, the art in Luminance is simple. The user interface is often a resource that can be tweaked to no end in the pursuit of ease of use. Most of the screen space will display the level itself with a band available for a toolset, similar to SimCity. Tools will be dragged from the set to places in the level. Each tool will have a count showing how many of that type of tool remains free for that level. If this count is zero, then the tool will be dimmed to indicate it is unavailable. We feel that this is a very intuitive way of presenting the player with objects to interact with.

We want the player to feel energized by the beam of light; its design reflects this. The beam will have an attractive shine to it and give off a sense of motion. Different tools will interact with the light beam in their own ways. Some tools will simply change the direction of the light while others will try to highlight a filtering or prism effect. Most of Luminance's beauty will come from the light and its interactions.

All the tools available to the player will have distinct appearances to allow no question about how they operate. The focus is not to make them appear pretty; rather, useful. We want to emphasis that the tools exist to do a job.

Technical Design

Engine

To fully develop this game we split the project into two main pieces. The first piece is the framework for managing the game. This framework will be our game engine and it will be implemented as a C# library that an XNA game project can attach to and use. Ideally the framework will be written as generically as possible so to accommodate any game, including 2D and 3D components. The engine needs to be able to handle multiple game states and multiple in-game windows per state. Game states in our definition will include a particular state of the implemented game. Examples of game states would be title screen, menu and in-game. The engine should handle transitions between two states, allowing us to add fade effects or other transitional effects. A state is more or less a place to put a bunch of state specific initialization or shutdown code. Each state has an update() and draw() function that the engine calls at the appropriate times.

Though a state can be rendered on its own and a game could be implemented within a state the idea is to have states define windows. These are not traditional windows you see on your desktop, instead they are in-game windows. These windows are used to render sections of the game user interface in a controlled manner. It is doubtful our game will make heavy use of more than one or two windows such as the game window and in-game pause window (with faded game in the background).

Another important part about the framework is user input. Since user-input is best handled in a last-state-compared-to-current-state method this is all abstracted for the user of the engine. The idea is each input device has events for pressing a button, releasing a button, holding a button. The input device will determine which events to fire whenever it is asked to process its input.

The input events will be keyed by the button pressed and the owner of the event. This way a state can bind events from multiple windows to multiple buttons. Input events can return true to cancel the bubbling of the events or false to allow it to bubble. Examples of usage here might be having a particular in-game window open that handled mouse clicks like the inventory, clicking in this window, the state would do the collision detection, process the input for the inventory window which would prevent the mouse click from going through to the actual game window behind the inventory.

An important part about every engine is how it handles physics and we aim to to provide a decent framework that is not too restrictive. A unit is something that inherits from components and is added to windows. A unit provides a physics packet from the valid (old) state to the new state. All updates on a unit will save the valid state and attempt to update the new state. Each unit then is compared to a list of other units provided by the game, not the engine. The game implementation has to determine which units to test for collision against other units. Units are then responsible for returning a list of points to test for collision on. These points can be vertices or other collision objects like bounding boxes or spheres. The engine will provide all the basic collision methods we will need. When the engine finds a collision it will call a function provided by the unit class that handles the next step of collision un-related to physics. This would be things like activating a power-up, touching the final goal or other events that occur after collisions have been dealt with.

The final core element of the framework is handling resources like imported graphics, shaders, fonts, sound and music. The engine will provide an easy way to get at XNA's content class that is used to load pretty much anything you need. Sound effects and music are simple resources that will be loaded by the resource manager. Sound effects will be loaded like all other resources, keyed by a name. Playback of sound effects is a bit tricky as sound effects are buffered and only have one read/write pointer. So to handle multiple playbacks of the same sound effect resource we need to have the engine spawn a particular number of versions of the same sound effect allowing playback to pick the first non-playing version of the sound effect. Music playback will provide extended controls to looping, fading, merging to get seamless music transitions.

To implement the game using the engine we simply begin to inherit from the framework classes. A title state will be created, defining some 2D components allowing us to do a small game intro. A menu state will be defined to give our game that standard start, continue, option menu. These menus will all be windows within the menu state. We define a static option class used to configure the game settings and allowing global access from any class to view the options. A game state is defined to accommodate the actual game. Within this game state will be two windows, one window for the entire game field covering most of the screen. Another window placed below the game area will represent the toolbelt. The tool belt is the users list of objects he can place into our game world. It will be a

clickable or scroll able list of icons with a number indicating how many of this item is left to place in the world.

Level Implementation

The playing field is contained within a map class. This class contains two copies of the map: a basic non-renderable version which is the raw data used to create the renderable map, and the renderable map itself. The raw data consists of a 2 dimentional field of nodes (which contain the type of object, with a pointer to any object in the cell) defining playable spaces as well as pre-placed tools. An example would be the following map:

2 1 0 1 0 3
0 0 1 0 0 1
0 0 0 0 1 0

where 0 is a free playable space, 1 is a non-playable space, 2 is the light emitter and 3 is the goal. These values are indexes into an array containing all of the possible tools, which allows maps to also contain pre-placed tools. This design allows end-users to create their own maps using a text editor. The raw data for the map class can come from two places: The random level generator, or textfile loading code. The current code does not handle loading of map files at all yet, but the architecture we have used makes this trivial.

The renderable map is just a 1:1 visual representation of the raw data. It takes a fixed size playing field and divides it up into the appropriate number of equally sized square spaces to form the play grid. Each grid square can contain only one tool, or be blank. This makes both the visual rendering and the internal record keeping very simple. Each time the user places or removes a tool the raw data array is updated with the new value and the visual representation is updated with the appropriate renderable components.

This two layer approach grants a couple of benefits: ease of placement checks and easy light-beam collision detection. When the user tries to place a tool the map can do a simple check of the 2D array, which tells us not only whether the cell is blocked but also which object is in the cell. The light-beam collisions will be done using the tools themselves. Each tool will have a number of places on it where light will be emitted, and these will always fire as long as there is at least one incoming beam.

These beams will be XNA Ray classes extended to include drawing code. Doing this makes rendering the beams much easier and simpler to do because the collision detection rays can also be used to draw the lines.

Random Level Generator

The level generator is the heart of Luminance's replay value. It will be implemented by exploring possible states for a level to be in. The initial state will consist of an emitter and one or more goals, locations choosen at random. The generator will then try to solve the puzzle. Once a solution is found, an opponent algorithm will play devil's advocate and introduce obstacles to obstruct the solution. This back-and-forth pattern will be continued until the level reaches an ideal degree of difficulty. This will have to be discovered through trial and error. Due to the nature of the search, we feel that a modification of the minimax algorithm from game theory will provide the most effecient means of coming to a solution. As one can imagine, the branching factor of searching through all sorts of possibilities can simply be astronomical. Minimax will provide a way for us to cut off its search at various degrees, and with alpha-beta pruning implemented we will hopefully avoid certain branches altogether.

The problem at this point is to cut down on branches further still while avoiding levels that may appear essentially the same. Placing the emitter and deciding how many goals to have and where will largely help in this regard. Other measures to take may be to randomly decide on a subset of tools before the search begins and thus creating a much more narrow search tree.

Minimax will also provide a convenient way of varying difficulty amongst the levels. The crux of the algorithm relies on a state's utility function; a measure of how favourable a particular state is. By skewing this result Luminance can lean towards levels with more or fewer obstructions. This skew can then be adjusted by the player or perhaps increment itself over time to provide an increasingly difficult puzzle. Of course limits will have to be in place; if the level was full of obstructions for instance, then it might be perfectly clear which tools needed to be used in order to solve the puzzle.

Description of Proof of Concept

Our proof of concept consisted mainly of the engine and the map. The engine will make our lives much easier later because it is a nice, extensible system which makes designing the game rapid and easy to do. We had the beginnings of unit placement code, but we had no light drawing or collision detection. This made the proof of concept presentation a bit dull but we had to get the basics done before we could add in the playable components and we just did not have time to do this.

Timeline for Remaining Work

Many changes need to be made to bring Luminance from proof of concept to release. One of the most basic upgrades is the implementation of menus and a title screen. Also required are transition effects like fading when the player pauses, quits, or starts a game. This can be done by one team member within a week. Level effects such as a shader for the light to make it look full of life and definition of the models for the tools will require another week or two. Basic physics to control the light using the tools will require another week. Also, the design of the level screens needs to be improved and drag and drop events attached to the toolset. Lastly, having an image in the background so the level does not look like a crossword puzzle will make a nice touch. The goal is to make it look three dimensional even though the puzzle only requires two dimensions. This will be the icing on the cake for the game's presentation, though it is of lesser importance. The level generator also needs to be completed and integrated into the game engine. This will take a few weeks, but is definitly within the alloted time. While this is a lot to do, we definitely believe it is within reach.

Summary

As is evident, there is still a lot of work to be done. However, we do have a path to completion. Luminance will appeal to all sorts of people with its ease of use, replay value, eye pleasing design and gratification from solving a puzzle. We have no reason to believe that we can not finish this project in the alloted time.