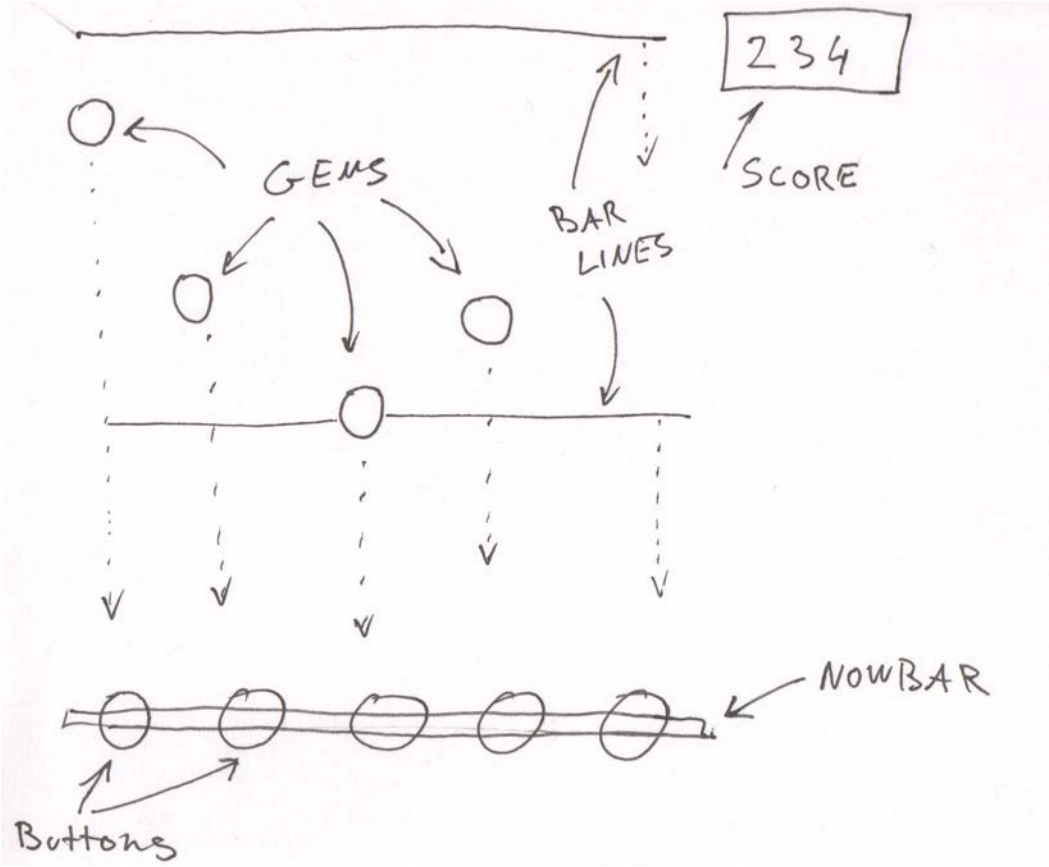
Assignment 7: Guitar Hero

Overview

In this assignment, you will make a simplified Guitar Hero game, focusing on the core game mechanic (without the background graphics / characters, etc...). The main simplification is that we will use the computer keyboard as input device instead of a plastic guitar. Playing a note is therefore simply a matter of pressing the right keyboard key at the right time, as opposed to holding down the correct fret button(s), and strumming at the right time.

Since this assignment is fairly involved, you are advised to start early.

Specification



Hit a key ('p') to start/toggle the song playing – this will start both the background audio and the guitar-solo audio.

Visual Display

As the song plays, *Gems* (visual display of notes which you will author to correspond to some notes of the guitar solo) will start "falling down" from the top of the screen and approach the

NowBar, located towards the bottom of the screen. Each Gem belongs to one of 5 *Lanes*. There are also 5 *Buttons (visual indicators),* attached to the NowBar. When a Gem reaches the NowBar, it will intersect with the corresponding Button of the Lane.

In addition to Gems, *BarLines* also "fall down" from the top of the screen. Each BarLine represents the beginning of a measure / bar in the music.

A *Score Display* in the top corner shows the player's score.

Player Actions

5 keyboard keys are mapped to activate the five Buttons.

The player attempts to "hit" the Gems by pressing the keyboard key associated with the Lane of a Gem when the Gem intersects the NowBar. There are three possible outcomes:

- *Hit*: The Gem was successfully hit. More specifically, a key was pressed when a Gem was within a time window centered on the NowBar and the key's Lane matches the Gem's Lane. The window is called the *Slop Window* and is +/- 100ms (ie, a total of 200ms in width)
- *Miss*: There are two miss conditions
    - *Temporal Miss*: A key was pressed, but no Gem was within the Slop Window.
    - *Lane Miss*: A key was pressed with a Gem inside the Slop Window, but in the wrong Lane. Any Gem that was Lane Missed is marked as unhittable.
- *Pass*: A gem passed a point in time (after passing the NowBar) where it can no longer be hit (ie, it is past the Slop Window).

Game Reactions

A well-designed game has direct visual and auditory feedback to inform the player about what's going on and how they are doing. The reactions for GH are:

Button Press: When a key is pressed and released, the associated Button's visual display changes accordingly.

Hit: The Gem shows that it was hit (for example, an explosion, flair, or some other exciting display). The Gem should then disappear or otherwise stop flowing down the screen to indicate it is no longer in play. The solo audio track unmutes. Points are earned.

Pass: The Gem changes visual state to show that it can no longer be hit. The solo audio track mutes.

Miss: A miss sound plays to indicate a missed note. A Lane Miss has the same visual effect as a Pass (it can no longer be hit).

Part 0: Choose a Song

I've put up a few songs that we used in the original Guitar Hero game. These are multitrack songs, where the main guitar part is split from the rest of the background part so that each song consists of two stereo audio files. Pick a song.

Part 1: Annotate the Song [10 pts]

Use Sonic Visualizer to annotate the gem locations in the solo part of the song. This is similar to what you did in Assignment 2, but you will be creating a Time Instants Layer. Each annotation is a time value and a text label.

Hints:

- When the song plays in Sonic Visualizer, hit ; (semicolon) to insert a new annotation.
- When you play back the song, you will hear a click at each annotation.
- Use the Edit tool to adjust these annotation locations so that they line up precisely with the guitar notes.
- To change the text label, hit E to bring up the annotations window.

Use the annotation labels to specify the location and type of each gem. Use whatever format or syntax you want.

Authoring the Gem data is a bit of an art form. The Gem pattern should be musical and follow the contours of the solo guitar line so it "feels right" to play it. You can choose how difficult you want the Gem pattern to be. You do not need to have a Gem match every note of the guitar solo. In the easy levels of Guitar Hero/Rock Band, Harmonix authored far fewer gems than guitar notes. Only the most difficult level had a (mostly) 1-1 correspondence between Gems and guitar notes.

Export the gems data file with "Export Annotation Layer".

As you did in Assignment 2, write code that parses this text file and store the data in the class *SongData.* You can do a rough initial pass to validate your code. Then in Part 3, you can refine your Gem authoring so it really feels right.

Use the same process to create BarLine data (i.e., an annotation on each downbeat) . You have the option of using the same annotation file (so it will have both bar data gem data), or two different files. In fact, it might be easier to lay down bar line data using the backing track.

Part 2: Basic Graphics and Audio [20 pts]

Create the graphical and audio elements of the game:

- **GemDisplay**: draws a single gem
- **ButtonDisplay**: draws a single button
- **BeatMatchDisplay**: draws Gems, BarLines, NowBar, and Buttons. Animates the Gems flowing down the screen (hint: create a Translate() object and scroll Gems and Barlines by animating trans.y).
- **AudioControl**: plays both background and solo audio files.

Instantiate the BeatMatchDisplay and the AudioControl. The Gems and BarLines should flow down the screen towards the NowBar as the song plays.

At the end of this part, nothing is interactive. Gems and BarLines simply flow down as the song plays (and they should pause if you pause the song).

Part 3: Interactive Elements and Game Logic [20 pts]

In this part, you will finish the game by adding all the interactive elements and audio/graphical Game Reactions.

- **GemDisplay**: implement functions to change the Gem's graphical look as needed: *on_hit*, *on_pass*, and *on_update* (to help with hit animation).
- **ButtonDisplay**: implement *on_down*, and *on_up*, to change graphical look when the user presses a key.
- **Player**: Create the game logic that implements the interactive behavior described in the Spec. Player receives button up/down events from MainWidget. It figures out what to do and calls functions on BeatMatchDisplay and AudioControl to make the appropriate reactions happen. It also keeps track of score.
- **BeatMatchDisplay**: implement the functions called by Player that affect visual display of Gems and Buttons.
- **AudioControl**: implement mute / unmute for the solo tracks, and play the miss-sound-effect with a WaveGenerator.
- **Score**: Show the score somewhere on screen.

Lastly, refine your Gem authoring so that it feels right and fun to play. If you are short on time, you need to not author gems for the entire song, but you should do at least the first minute. Of course, the more gems your author, the higher your score will be =)

Finally...

Provide a brief video of you playing your game and a README that explains how things work, especially if you have added additional elements to this pset. Please have good comments in your code. When submitting your solution, submit a zip file that has all the necessary files.

21M.385 Interactive Music Systems

Fall 2016