Gloss Image Simulation

Program Design and Data Structures, 20 c
PKD, Spring 2016, Uppsala University

Dennis Englund
Albin Jansson
Martin Sellergren

Contents

1 Introduction

We have made a graphics simulation using the gloss-library in Haskell. You select an image on your computer and start the application. Then you're presented with a mess of shapes of different colors. When you press play, each shape moves towards a designated location, they may collide with each other, and eventually you can see an image emerging. Together the shapes form a version of the image you selected in the beginning.

We call these shapes objects. The program splits the image into different quadratic parts, each of which becomes an object with properties depending on the image part it represents. The shape and texture of the objects is determined by input arguments to the program.

2 User manual

2.1 Software requirements

To use this program you must install a Haskell compiler. Once you have the compiler installed the next thing to do is to install Gloss by typing **cabal install Gloss** in the command prompt(cmd). You also have to install JuicyPixels by typing **cabal install JuicyPixels** and gloss-juicy by typing **cabal install gloss-juicy**. The recommended versions are gloss 1.9.4.1, JuicyPixels 3.2.7 and gloss-juicy 0.2.

2.2 User Guide

To compile the program open a terminal, go to correct directory and type:

ghc main -threaded. The -threaded argument is important for the simulation to run smoothly. Then, to run the program type **main imgPath objectsOnWidth objectMode startMode accConstant frictionConstant collisions objectSizeFactor**

This may look daunting but I'll describe each argument below.

imgPath - Path to image to use in simulation, on the form a/b/c.jpg. Any common image format may work.

objectsOnWidth - Number of objects on one horizontal row to split the image into. An integer larger than zero. Lag may occur at higher numbers.

objectMode - Specifies the shapes and textures for the objects. The shape may be a ball or a square. The texture may be solid or vivid, where solid means single colored and vivid means multicolored. There are 4 different options, represented by numbers 1-4:
1 = Solid balls
2 = Solid squares
3 = Vivid balls
4 = Vivid squares

startMode - Start position mode for the objects; RandomSquare or Done. RandomSquare means objects are randomly positioned in a square and Done means simulation is done from start.

accConstant – A real number that specifies how fast an object's acceleration increases with distance from its target. Recommended: 0.1 - 2.

frictionConstant – A real number that specifies the amount of friction, i.e. the norm of the friction vector for the objects. Recommended: 100.
(note: increase frictionConstant AND accConstant to make simulation complete faster.)

collisions - Turns collisions on or off. Represented as True (on) or False (off). If the shape of the objects is square there will be no collisions.

objectSizeFactor - A real value between 0 and 1 where 1 means no padding around objects in completed animation, and 0 means no size.

There will be an upper limit for objectsOnWidth depending on your computer performance. If you exceed this limit the simulation will lag and collisions won't work as expected. Object's speeds won't be calculated correctly and the simulation may not finish. For my computer, a somewhat modern laptop I find that this upper limit is around 30 for simulation without collisions and 10 with collisions, for a quadratic image (with a total of objectsOnWidth*objectsOnWidth objects). This upper limit will also depend on accConstant and frictionConstant.

Don't be alarmed if you see a black screen for some time after you've started the program. The loading time will sometimes be quite long, especially if you've specified a large image. To reduce loading time, use images in bmp-format.

When the loading is done you will have to press P to star object movements. Press P again to pause. You may zoom out and in using Z and X, rotate view using A and S, and reset to original zoom and rotation by pressing R. Press Esc to quit the program.

3 Program documentation

3.1 External packages

The program uses three external packages; the gloss package, the JuicyPixels package and the gloss-juicy package. The gloss package is what our program relies on most heavily. This package provides us with a game loop, the means to construct graphical elements, the means to handle events, screen rendering and more. We use the JuicyPixels package to load an image from file and extract the required information from it to create objects. The gloss-juicy package has a function to convert a JuicyPixel's image into a gloss bitmap image which we use.

3.2 Data structures

There are six data types created and used in the program. Here is a brief explanation of them:

World = W Objects ViewPort Bool
A world with objects os, view port vs (for zoom and rotation) and pause flag p is given by: W os vp p. The world is holding all the information about the simulation at a given time. This is an abstract data type with the following interface:

World – for type signatures
initWorld – construction of an initial world based on input arguments
renderWorld – render world to a gloss picture
stepWorld – applies changes that time brought on a world
handler – changes a world based on user input

Object = O Picture Float (Float,Float) (Float,Float) (Float,Float)
Object of picture pic, side length(dim) d, position (x,y) speed (vx,vy) and target position (tx,ty) is given by O pic d (x,y) (vx,vy) (tx,ty). Target position is the position where the object is supposed to end up when the simulation is done. This is a hidden data type, used in the abstract data type Objects.

Objects = Os Shape [Object]
A list of objects os where every object has shape s is given by Os s os. This is an abstract data type with the following interface:

Objects – for type signatures

initObjects – construction of an initial list of objects based on input arguments
setStartPositions – set start positions of every object in list according to the start position mode
renderObjects – render objects to a gloss picture
stepObjects – applies changes that time brought on the list of objects

Shape = Ball | Square
Represents the shape of an Object which can either be Ball or Square.

Texture = Solid | Vivid
Represents the texture of an Object such that Solid is single colored and Vivid allows different colors for each pixel.

StartPositionMode = RandomSquare | Done
Defines different starting constellations of objects. RandomSquare means randomly positioned in a square. Done means start positions are same as target positions and simulation is therefor done from start.

3.3 Functions and algorithms

This section starts with the functional specifications for the main elements in the program followed by a description of the control flow, and finally a more overall description of the program's main functionalities.

3.3.1 Functional specifications for main elements

main
PURPOSE: Start the simulation.
SIDE EFFECTS: Reads input arguments, reads image from hard drive, outputs graphics based on input.

initWorld ww wh img shape texture objectsOnWidth objectSizeFactor animationOfWindow seedNumber startPositionMode
PURPOSE: Get initial world.
PRE: ww, wh and animationOfWindow >= 0, objectsOnWidth > 0, 0 <= objectSizeFactor <= 1 (note: objectsOnWidth also determines the size of the objects. If n is low and the picture's ratio (width/height) > 1, no objects may fit on the height and therefor no objects will be created.)
POST: The initial world based on the input arguments.
initObjects ww wh img shape texture objectsOnWidth objectSize-

Factor animationOfWindow
PURPOSE: Get initial list of objects.
PRE: ww, wh and animationOfWindow >= 0, objectsOnWidth > 0, 0 <= objectSizeFactor <= 1
POST: img transformed into objects of specified shape and texture, with dimensions and target positions according to objectsOnWidth (=number of objects on screen width), window dim (ww*wh) and objectSizeFactor, positioned at (0,0) and with no speed.

setStartPositions os sn mode
PURPOSE: Set start position for objects.
PRE: True
POST: Each object in os set to start position according to start position mode, with possible help of seed number sn.

renderWorld world
PURPOSE: Render world.
PRE: True
POST: World rendered into picture.

renderObjects os
PURPOSE: Render objects.
PRE: True
POST: Picture containing every object's picture in os, where each objects picture is positioned according to its position (x,y).

stepWorld aC fC collisions t world
PURPOSE: Step world one time step.
PRE: aC, fC and t >= 0. (Note: Keep aC and t small for correct collisions.)
POST: world after one time step of length t seconds, where acceleration and friction is determined by aC and fC, with object collisions if collisions is True. Step not taken if pause flag of world is True.

stepObject t os aC fC collisions
PURPOSE: Step objects one time step.
PRE: aC, fC and t >= 0
POST: The position and velocity of each object in os updated based on amount of time that passed since last step i.e t seconds.
New position is equal to old position plus old velocity multiplied by t.
New velocity is equal to old velocity plus acceleration which is a function of distance from center and friction. New velocity is zero if object is sufficiently close to its target while moving slowly.

Acceleration is determined by acceleration constant aC and friction constant fC where aC determines how fast an object's acceleration increases with distance from its target, and fC is the amount of friction, i.e. the norm of the friction vector. An object's acceleration is the sum of a pull-vector directed towards the target and a friction-vector in opposite direction of the velocity. If collisions is True, the velocities of the new objects will be adjusted according to internal collisions. No collisions for squares.

handler event world
PURPOSE: Handle events.
PRE: True
POST: World with changes determined by event. Pause flag and viewport (zoom and rotation) of world is the changeable attributes.

3.3.2 Control flow

Using all the functions described above including their helper functions in combination with imported functions, the program runs. Firstly, main is evoked to start the program. In main, the gloss function 'play' is called which is what creates the simulation basically.

Play takes arguments that describe the simulation – a display-object for basic window appearance, a background color, number of steps to take per second by the animation, an initial world, a function to render a world to a picture, a handler-function to handle the user inputs, and finally a function to step the world one time step. All rendering to screen and such mechanisms is handled under the surface by gloss. Much of the inspiration on how to make animations in Gloss and on using the play-function is taken from a Gloss Pong-tutorial[3].

Before calling play, main will read input arguments by getArgs, read the specified image from hard drive and also create a random integer using randomIO to be used as a seed number for the randomness during initialization of object positions. main is the only function in the program that uses the do-notation.

The play function in main particularly requires an initial world, means to render this world, means to step this world in time, and a user input handler. This is what our code mainly provides and it's described in detail in the following section.

3.3.3 Description of main functionalities

Initial world: The function for initializing the world is initWorld with specifications above. Based on input, a new world is created. This new world has a default view port (for zoom and rotation) and a pause flag set to True. All new objects of this world are created in initObjects and they're set to initial positions in setStartPositions. For creating a new object, a function called createObject is used. This function creates an object with given shape and texture. createObject makes sure that the finished animation picture fits inside the screen (with adjustments by the screenSizeFactor input argument) when calculating the target position for the new object. Target position and dimensions is calculated based on the position and size of the image box the new object is based on inside the provide image.

The functions for giving texture to an object (actually it supplies a Picture with a texture to the object) are in different ways extracting colors from an image and applies them to a Picture. A function called avgBoxColor is used to create a monochrome texture by calculating the average color in an area of the provided image, and creates a Picture with this color.

A polychrome texture is made using a function called createVividPic combined with drawBall or drawSquare depending on the shape. These combined functions gives the object a picture with texture according to the pixels in an area in the image. Inspiration on how to extract RGB colors from images is taken from a thread on Stackoverflow[1].

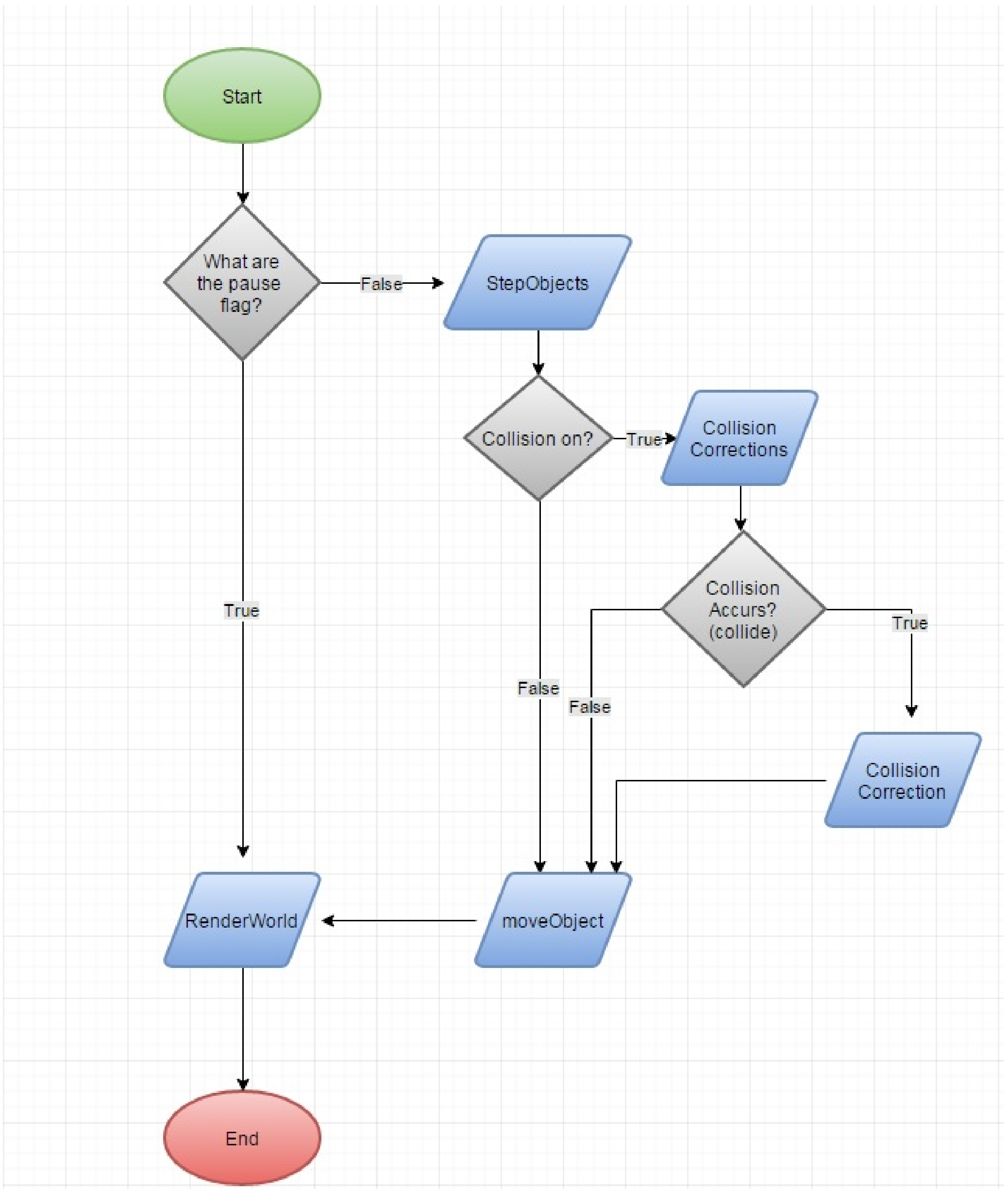
Render world: Rendering the world is done in renderWorld, which applies the view port of the world to the picture created by the function renderObjects. All display in the simulation is the objects. renderObjects will simply go through the whole list of objects in the world and return a picture of every object translated in the x-y-plane based on its position, and combine all these pictures into one. This is simply implemented using the gloss functionality.

Step world: Stepping the world one time step is done in stepWorld. This function will check the pause flag of the world and if it's false it will call stepObjects to get a new list of all the objects where each object has updated positions and velocities. stepObjects does two things – moves objects and changes velocities by a function called moveObject, and then corrects the velocities based on internal collisions in a function called collisionCorrections. What moveObject does is described in the specifications for stepObjects above, and a description of collisionCorrections is provided below.

Collision corrections: The function that handles collisions is called collisionCorrections. The algorithm is as follows:
-*start* with first object in list
-check if it collides with any other object in the list by iterating through the other objects. Only look for the first collision that may happen while iterating. If collision happens - update involved objects' velocities and put these objects first in corrected-objects-list. If there is no collision just put the first element first in corrected-objects-list. Concatenate corrected-objects-list with the list: "go to *start* with object list without these two or one corrected objects". (Repeat recursively). When no objects when beginning from *start* - return empty list. Then you have a list with the objects with updated velocities.

'Whether two objects collide or not' is evaluated in a function called 'collide'. For balls, it is checking if the distance between the two objects' center points is less than the diameter of a ball and if the objects are getting closer to each other. If both these conditions are satisfied then it is a collision.
The calculations we used for new velocities of two balls after a collision has happened is described in the following article: 2-dimensional elastic collisions without trigonometry[2] and these calculations are done in a function called collisionCorrection. When slow collisions happen between two balls, there is an applied speed boost so that the balls don't get stuck so easily.

Handle events: The function that handles events is 'handler' described above. This function will change the world according to its input, i.e. information about which key on the computer that is pressed or released. If one of the zoom or rotate keys is pressed, handler will change the view port of the world. If P is pressed, handle will switch the pause flag p to 'not p'.



A heavily simplified flowchart describing an Object in stepWorld.

4 Shortcomings

One shortcoming is the big performance impact of collisions. This is a big limitation for the number of objects possible without too much lag. A possible method to improve this might be to split the screen into sections, determine which objects are in which section and only check for collisions between objects that are in the same section. Also, collisions when three balls are involved (likely to happen because of the discrete time steps) are handled like an ordinary ball to ball collision, and the third ball is ignored. Furthermore, collision detection for squares isn't implemented.

It was also harder than expected to do test cases for some of the functions, especially those which answer depends on the color/colors in some place in an image. Then you typically have to know the color/colors in that area which is a hard thing to take out by hand. One more thing that was hard if not impossible to test is functions with IO. (To test the test cases we've written, just type performTests in ghci when main.hs is loaded.)

5 Summary

This project was about creating a visual image simulation in the Haskell library Gloss. In the finished program you select an image with a common type on your computer, choose some arguments about the simulation, and then run the simulation. You are then presented with a window with shapes of different colors. The shapes are aligned in a rectangle in the middle of the screen and when you press p every shape starts moving to its designated location. The shapes may collide and eventually you can see the image you chose emerging of the shapes when they reach their designated positions. There are some choosable simulation options which can be read about in 2.2 User Guide.

Overall the project went well despite the fact that group Member Martin wrote much of the code except test cases and also did much of the documentation of the project. He did this very well but in a group project every member should participate equally much. This can be explained by several reasons, one is that the other two started later with working on the project and when they tried to learn basic functions in Gloss the code was already finished, they should have started earlier. Also that Martin put down most time of us on the project.

One more reason can be that Martin is at the moment a better programmer than the other two and really got into the programming part. The teamwork and communication in the project group could also have been better earlier in the project and there we could have somehow divided the workload more equally. It seemed easier for Martin to write much of the documentation because he knew the code best of us. The other two spend much of their project time on understanding the code but also on doing documentation and test cases for the code. All-in-all it has been a good learning experience, since most of us haven't really done anything like this before, and we will take the things we have learned over to another project in the future.

References

[1] A thread on stackoverflow talking about extracting rbg values from a picture. *Accessed 2016-03* `http://stackoverflow.com/questions/31883755/how-to-extract-rgb-values-from-most-pictures`

[2] 2-Dimensional Elastic Collisions without Trigonometry. *Author: Chad Berchek, 2009. Accessed : 2016-03* `http://www.vobarian.com/collisions/2dcollisions2.pdf`

[3] Your First Haskell Application (with Gloss) *Author: Andrew Gibiansky, 2014. Accessed: 2016-03* `http://andrew.gibiansky.com/blog/haskell/haskell-gloss/`