Group Members: Justin Peng (jfp2130), Akash Nayar (akn2120)

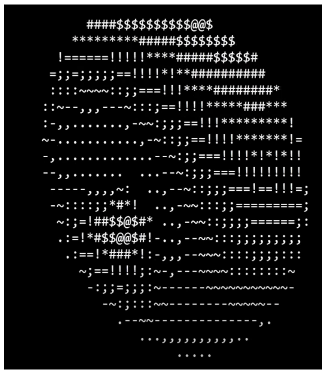
Project Name: Howard

Our idea for the final project is to create a two-dimensional ray-casting simulation. The project will use the terminal to display a birds-eye view of a configurable 2D environment. This environment is essentially a matrix consisting of 0's (empty space) or 1's (walls). The user will then be able to place multiple light sources throughout the environment and watch the light appropriately spread and fill out the available space.

Light rendering algorithms have seen increasing popularity in recent years. Many methods such as ray tracing, simulate individual "rays" of light, tracking their movement through space and how they interact with objects. The propagation of rays is simulated via time-stepping methods, meaning that each ray travels through the scene in incremental steps, and information about the ray is updated as it continues propagating throughout the scene. In a practical use case, millions of rays are propagated throughout the scene and their interaction with the environment is tracked simultaneously. Furthermore, attributes like a ray's current brightness, or its index of refraction require relatively intensive calculation. Thus, light rendering algorithms often require powerful hardware.

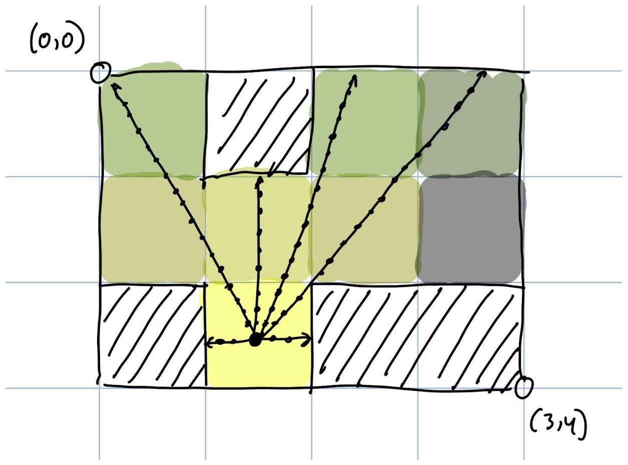
Light ray propagation presents an interesting candidate for parallelization. The nature of individually simulated light rays and time-stepping methods allow us to subdivide the work of the simulation between threads. It is possible to subdivide the light rays in the scene between threads or mark the light rays emitting from specific light sources to be handled by a specific thread.

We plan to use the terminal window as our user interface. Space characters will represent empty space, 'X' characters will represent walls, and 'O' characters will represent a light source. Similar to the donut on the right, characters such as '#', '$', '!', and '=' will denote varying levels of light intensity.



```
      ####$$$$$$$$$$$@@$
    *********#####$$$$$$$$
  !======!!!!!****#####$$$$$#
 =;;=;;;;;;==!!!!*!**##########
 ::::~~~~::;;===!!!****#########*
::~--,,,---~:::;==!!!!*****###***
:-,,........,-~~:;;;==!!!*********!
~-............,-~::;;==!!!!*******!=
-,..............--~:;;===!!!!*!*!*!!
--,,.......   ...--~:;;;===!!!!!!!!!
 -----,,,,~:  ..,--~::;;;===!==!!!=;
 -~::::;;*#*!  ..,-~~:::;;=========;
  ~:;=!##$$@$#* ..,-~~::;;;;======;:
  .:=!*#$$@@$#!-..,--~~:::;;;;;;;;;
   .:==!*####*!:-,,,--~~~::::;;;;:::
     ~;==!!!!;:~-,---~~~~::::::::~
      -:;;=;;;:~------~~~~~~~~~~~-
        -~:;:::~~---------~~~~~--
          .--~~-------------,.
              ...,,,,,,,,,,,,..
                   .....
```

We will create vectors stemming from our light source at regular intervals of angles and propagate them in small increments (perhaps 1/10th of the grid granularity), as seen in the diagram on the right. We can represent a light source by creating a type in Haskell, storing the position, and luminosity of the light. We can represent a ray by creating a type in Haskell, storing the current angle, position, the originating light source, and brightness of the ray. The propagation could be done using list comprehensions or mapping in Haskell: For a given ray with unit direction $< x, y >$, we begin at the light source. We increment the position of the ray by $c \cdot < x, y >$, where c is some small constant. The ray's new position is $< (1 + c) \cdot x, (1 + c) \cdot y >$. We then update the brightness of the current cell based on the ray's new state. We continuously update the list until each ray's encompassing grid block is either a wall or outside the bounds of the environment. To account for collision with a wall, we check the current grid element of the ray. If that element corresponds to a wall, or the ray's current position is within a certain distance of a wall, then our ray has intersected with an object and should thus be terminated. The light will have a brightness calculated using the equation $b = L/d^2$, where L is the luminosity of the light source and d is the distance from the light source. Users will be able to move a light source using the WASD keys and place it in position using the enter key. Once this happens, the current light source will be locked in place and a new one will be added in to be moved around by the user.



We plan to parallelize the light calculations for each different light source by assigning a light source to an individual thread. Thus, adding in more light sources should not increase the render time of each frame (at least until we exhaust all the cores on the machine). Alternatively, we may also explore subdividing groups of rays to be handled by different threads, by having a thread continuously update its group of rays until they are all terminated.