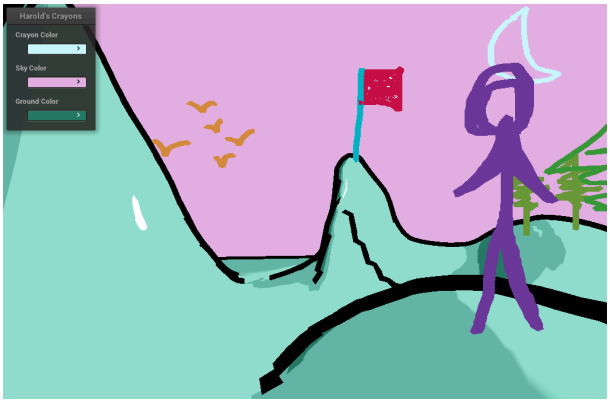
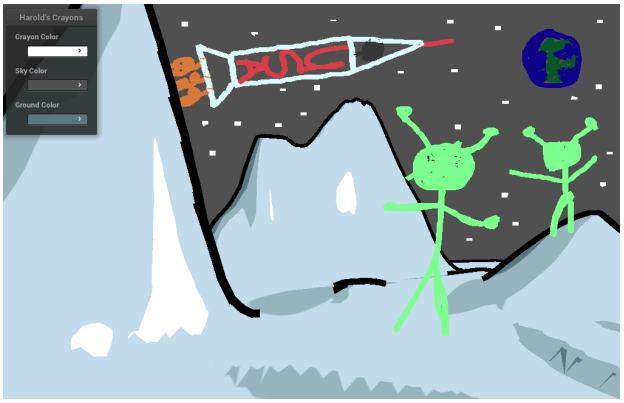
CSCI 4611: Programming Interactive Graphics and Games

Assignment 6: Harold

Handed out: Mon, 4/23

Due: Tue, 5/1 11:59pm





This culminating assignment in the course is inspired by the 1955 children's book, Harold and the Purple Crayon by Crockett Johnson. Harold is a little boy who creates his own virtual worlds just by drawing them with his magical purple crayon. In this assignment you'll bring Harold's magic to your computer screen.

In addition to inspiring children for decades, Harold can also claim to have inspired some exciting early computer graphics research in non-photorealistic rendering. In this assignment, we will be implemented portions of the paper, "Harold: A World Made of Drawings", presented by Cohen et al. at ACM NPAR 2000, the 1st International Symposium on Non-Photorealistic Animation and Rendering. If you like, you can watch a video of the original system. The lead author of this paper, Jonathan Cohen, was about your age (a junior or senior in college) when he developed this system and published this as his first research paper. He then went on to work on movie special effects at Industrial Light and Magic. This is the kind of thing you may also be able to do if you continue studying computer graphics and get involved in the computer graphics, visualization, and VR research groups at the UMN!

In this assignment, you should learn:

- How to perform 3D mesh editing operations in response to user input.
- How to use pick rays and intersection tests to convert 2D screen space coordinates (e.g., the mouse position, a 2D curve onto the screen) into a 3D virtual world.
- How to implement a computer graphics algorithm described in a research paper.

- How to analyze and integrate new code into a complete interactive program written in C++.

Requirements

Background

Please note that the time to complete this assignment is a bit shorter than typical. We have accordingly reduced the requirements to fit the time allotted. The three main features in Harold are: 1. Drawing hills and valleys on the ground. 2. Drawing strokes in the sky. 3. Drawing billboards (including adding multiple strokes to the same billboard). Instead of implementing all three, you will just be implementing #1 and #2. The support code already contains a complete implementation of #3 for you.

Harold is a complete application, so there is quite a bit of support code provided for this assignment. Start by reading through the code, which is heavily commented, and learn how a computer graphics program like this is put together.

The **HaroldApp** class is the main application class and implements the sketch-based user interaction techniques. One of the things that is really cool about Herald is that the user input can be interpreted differently based upon the current content. The original Herald paper did include some drawing modes in order to support some extra features, but for our version, we do not need any buttons to turn on “draw on the sky mode” vs. “draw on the ground mode” vs. “create a billboard mode” – we just draw, and the system figures out our intent. This table shows how the properties of the path drawn by the user’s mouse (called the “stroke”) are used to trigger different 3D modeling operations:

Stroke Made by Mouse	3D Modeling Operation
Starts in the Sky	Add a new stroke to the sky
Starts AND ends on the ground	Edit the ground mesh to create hills and valleys
Starts on the ground and ends in the sky	Create a new billboard
Starts on an existing billboard	Add the stroke to the existing billboard

As the user draws a stroke the HaroldApp class records the 2D path of the mouse as a series of 2D points, samples along the path. This is stored in the **stroke2d_** array, and you’ll need to work with this array, which is of type **std::vector<Point2>** to implement some of the features.

We can't draw the **stroke2d_** array directly to the screen – it needs to be converted into triangles first. So, the **HaroldApp** class also maintains a **stroke2d_mesh_** that is a simple 2D triangle strip. The triangle strip follows the centerline of the **stroke2d_**, adding just a bit of thickness around it so that it can show up nicely on the screen. You'll also need to work with this triangle mesh to implement some of the features.

Please note, the 2D points and vertices stored in **stroke2d_** and **stroke2d_mesh_** are not in units of "pixels"; they are in "normalized device coordinates", where the top left corner of the screen is (-1,1) and the bottom right is (1,-1). This is convenient because this is the coordinate system that is needed when converting a point on the film plane to a point in 3D space, such as with **GfxMath::ScreenToNearPlane()**. As explained in class, this is a key step in creating a pick ray.

Requirement 1: Drawing in the Sky

It is easier to implement Drawing in the Sky than editing the ground mesh, so we recommend that you start here. All of the Sky portions of the program are implemented in the **Sky** class (sky.h and sky.cc), and the support code already has a structure in place for calling **Sky** when appropriate in order to create new strokes and draw them. The **Draw()** call is already completely implemented for you. Your job is simply to implement the **Sky::AddSkyStroke()** function.

This function takes the **stroke2d_mesh** described earlier as input. We want to project this mesh onto the sky, which is really a huge sphere of radius 1500.0. To do this, you need to create a 3D mesh that has the same structural connectivity as the **stroke2d_mesh** but that has vertices that lie on the sky sphere rather than on the 2D screen. There are several ways to do this, but we think the easiest is to start by copying the **stroke2d_mesh** into a new mesh. You can do this by simply creating a new mesh and setting it equal to **stroke2d_mesh** (e.g., **Mesh m = stroke2d_mesh;**). The new mesh will then have the exact same vertices and indices as the original. We want to keep the same indices, but the vertices need to be altered. So, your next step should be to loop through the vertices in the new mesh and convert them to 3D points that lie on the sky sphere. The **Sky::ScreenPtToSky()** function will be very helpful for this.

When you're done, store the result in a new **SkyStoke** struct. (See sky.h for the definition of **SkyStroke**.) Don't forget to store the current stroke color there as well.

This portion of the assignment should not be too difficult. Completing this corresponds to a 'C' level effort.

Requirement 2: Editing the Ground

The second requirement is to edit the ground, which you should do in the **Ground** class (defined in ground.h and ground.cc). A **ground_mesh_** is already created for you in the support code. It's a very simple geometry, just a regular grid of triangles, sort of like a checkerboard where each square is divided into 2 triangles. To simplify the mesh-

editing algorithm, we will not do any adding or subtracting of vertices. All we will do is move their positions up or down to create hills and valleys. Note that this will mean that some of the triangles will be really stretched out, and you'll probably notice some lighting artifacts due to this, but this isn't too distracting since we are using a sketchy rendering style anyway.

The specification for how to edit the ground in response to the stroke drawn by the user comes from the Harold research paper. We'll follow the algorithm and equations described in Section 4.5 of the paper, which is quoted here.

> Terrain-editing strokes must start and end on the ground. Call the starting and ending points *S* and *E*… [T]hese two points, together with the **y**-vector, determine a plane in R^3, that we call the *projection plane*. The points of the terrain-editing stroke are projected onto this plane (this projection, which is a curve in R^3, is called the *silhouette curve*); the shadow of the resulting curve (as cast by a sun directly overhead) is a path on the ground (we call this the *shadow*). Points near the shadow have their elevation altered by a rule: each point *P* near the shadow computes its new height (*y*-value), P'_y, as a convex combination[1]
>
> $$P'_y = \begin{cases} (1 - w(d)) \cdot P_y + w(d) \cdot h & \text{if } h \neq 0 \\ P_y & \text{if } h = 0 \end{cases}$$
>
> where d is the distance from P to the projection plane, h is the y-value of the silhouette curve over the nearest point on the projection plane to P, and $w(d)$ is a weighting function given by
>
> $$w(d) = \max\left(0, 1 - \left(\frac{d}{5}\right)^2\right).$$
>
> This gives a parabolic cross-section of width 10 for a curve drawn over level terrain. Other choices for w would yield hills with different shapes that might be more intuitive, but this particular choice gives reasonable results in most cases.
>
> Note that if the silhouette curve bends back on itself (i.e. it defines a silhouette that cannot be modeled using a heightfield), then the variation of height along the shadow will be discontinuous. The resulting terrain then may have unexpected features.

The h in the equations is a bit complex to calculate, so we have provided a function called **hfunc()** that you can use to calculate h given, as the text describes, the silhouette curve, the closest point on the projection plane to the vertex we are editing, and the normal of the projection plane.

---

[1] This equation is edited slightly from the original text to include the two cases.

The support code also includes some comments to help your organize your code to implement the 3 main steps of the algorithm:

1. Defining the projection plane.
2. Projecting the 2D stroke on the screen into the projection plane to create the silhouette curve.
3. Looping through the vertices of the ground mesh and adjusting the y-value of each vertex according to the equations above.

In addition to the **hfunc()** function, you'll also find the **ScreenPtToGround()** function quite helpful. Also, don't forget about the MinGfx documentation. The **Point3** class, for example, provides some useful functions, such as finding the closest point on a plane.

This portion of the assignment is more challenging. A partial implementation corresponds to a 'B' level effort and a complete working implementation corresponds to an 'A' level effort.

Above and Beyond

All the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work — if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

We will not implement the complete Harold paper in this assignment, so one of the most obvious extensions is to go back to that paper and implement some additional features, such as "bridge strokes", "ground strokes", or "navigation strokes". Or, you could write some different stroke shaders to implement watercolor brush strokes. It would also be neat to see these drawing somehow "come alive"! Not sure exactly what that means, but use your imagination!

If your ideas for going beyond the requirements would make your code more difficult for the TAs to grade, please help them out by submitting a standard version of your assignment first through the normal website link, and then email the TAs the fancier version of your assignment.

Support Code

You should *not* need to update MinGfx in order to complete this assignment. The update you did before starting assignment 3 should be the last one needed for the course.

As in past assignments, you will need to download the support code for assignment 6, which is posted on canvas as a zip file. Download the support code and place it in a directory named a6-harold within your dev directory. Then do:

```
cd dev/a6-harold

mkdir build

cd build

cmake-gui ..
```

“Configure” then “Generate” then “Open Project”

Handing It In

To hand in your code, check in to the master branch of your Github repository by the deadline. Any commits past the deadline will be assessed according to the late penalties described in the syllabus.

Further Reading

Check the hyperlinks in the intro to this document to link directly to the original Harold paper and video.