Hue Clue

Team:

Amanda Sparks

Lauren Wang

Lana Glisic

Live demo: https://amanda-2.github.io/hueclue.github.io/

Abstract

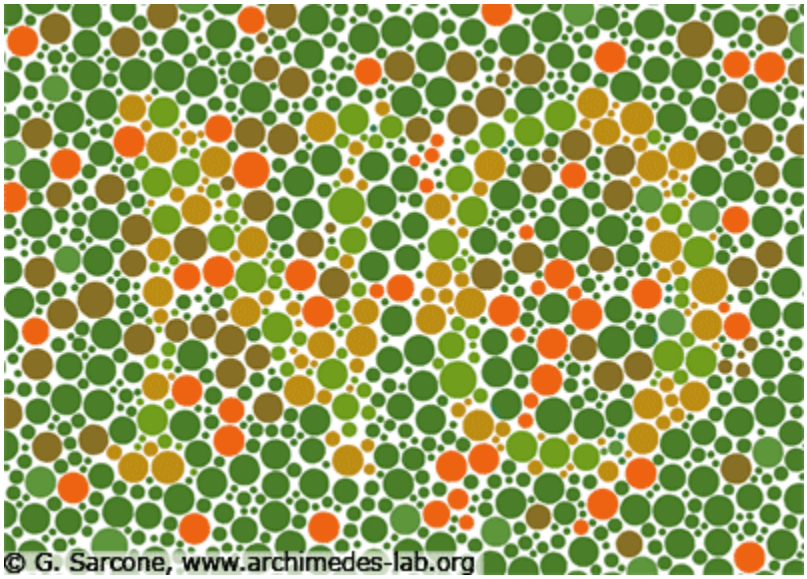
This paper describes the production and development of Hue Clue. Hue Clue is a game that challenges the player's ability to differentiate between different colors. As the game progresses, the difference between colors becomes smaller, and more differences are included to increase the difficulty of the game. When the user quits the game, they are given information about their score and what they achieved during their playthrough. This was accomplished using Javascript and the ThreeJS libraries.

Introduction

The goal of this project is to create an engaging game that challenges players to test the limits of their color vision. Hue Clue is inspired by comparative color perception in humans as well as colorful optical illusions. While playing, players can assess their ability to spot subtler and subtler color shifts among the displayed objects. This requires attention to detail and careful analysis of the three-dimensional scene. Lighting, occlusion, and perspective in the scene can warp color perception in unexpected ways. This requires the player to rotate around and zoom into the scene to get a better look. Players can train themselves to spot finer differences in color over time, and the game can be played solitarily as an exercise in focus and mindfulness. Upon ending the game, the user is provided with statistics describing metrics like player speed and the degree of perceptual difficulty that the player has overcome. These metrics provide an extra dimension to the game, allowing players to compare statistics with others and attempt to break their own records. This facilitates engagement for both personal satisfaction and friendly competition, benefitting multiple kinds of players.

A mobile game called *I Love Hue* focuses on subtle color differences. The developer website instructs the user to “reorder mosaics of coloured tiles into perfectly ordered spectrums”, challenging the user’s ability to differentiate between different hues and shades of colors, as well as organize them. The game displays a grid of colored squares, and the user taps on the square they want to move. Each colored tile touches and there are no shadows or rotational abilities.

An inspiration for this game was the traditional colorblind tests and the reverse color tests. The traditional colorblind tests use a number of colored dots, arranged in a pattern that displays a certain image to people without colorblindness and a different image or no image to those with colorblindness. (insert images below). Reverse colorblind tests work in the opposite manner, where only those who experience color blindness are able to clearly see the image within the pattern.



Approach and Methodology

Scene - Lauren Wang

Scene

The `SeedScene.js` script initializes and updates the game state, including the level number and game statistics (score, completion time, and offsets), and adds components such as lights, a floor, a skybox, and 3D primitives to the viewport. When the level progresses, `app.js` calls a new SeedScene to regenerate the next level.

Primitives

In the `primitives.js` script we implemented a Primitive class that manages the creation, positioning, and customization of 3D shapes in the scene. Its primary goal is to generate a set of 3D objects with specific characteristics, ensuring they are positioned without collision, and apply procedural colors or textures dynamically.

Initialization
The constructor initializes a group of primitives with properties passed from the scene, such as the number of primitives, color offset, shape of the primitives, and textures. The class then defines key parameters, such as the initial size, minimum size, spacing between primitives, and spawn radius, for generating placements.

Primitive Types
The script supports multiple types of primitives by dynamically switching between geometric shapes, including cubes, spheres, cylinders, and cones. We use THREE's built-in geometry functions to create these shapes. Each primitive type has a specific size, y-position, and geometry logic. For example, cubes are aligned on the floor using their half-size, while spheres are centered based on their radius. This flexibility makes the class adaptable to diverse scenes and use cases.

Placement Algorithm & Dynamic Resizing
A key feature of the implementation is the use of *Spatial Hashing* to ensure collision-free placement of primitives. The spatial hashing algorithm divides the space into grid cells, where the size is determined by the size of the primitives plus spacing. This ensures that any two primitives occupying different grid cells are guaranteed not to overlap. For each primitive, a unique hash key is calculated using the primitive's coordinates. Before placing a new primitive, the algorithm checks the hash map for neighboring cells to determine if any existing primitives might collide with the new one. If a collision is detected, the algorithm retries placement by selecting a new random position within the spawn radius. It continues this process until the primitive is successfully placed or the maximum number of attempts is reached. If the algorithm cannot place all primitives without collisions after maximum attempts, it reduces the size of the primitives and retries the placement process. This iterative resizing ensures all primitives can fit within the given space, or logs a warning if placement becomes

impossible. Once a primitive is successfully placed, its hash key is stored in the spatial hash map along with its position. This map acts as a registry of occupied grid cells, enabling rapid lookup during subsequent placement checks. After successfully placing all primitives, the meshes are created and their orientation are also randomized to add variety and dynamism to the scene. The advantages of the spatial hashing algorithm lie in its efficiency and scalability. A limitation of this method is that, due to random positioning and resizing, the final size of the primitives may vary inconsistently as their numbers increase significantly.

Textures and Materials

Three types of procedural textures—checkerboard, stripes, and zebra patterns—are applied as the level of difficulty increases. If no texture is specified, a default material with a single color is applied. The three textures are dynamically created using canvas element in a separate script called `textures.js`.The textures are customizable with user-defined colors, grid sizes, or stripe orientations and are applied to MeshStandardMaterial.

Sprite Labels

The script also includes floating text labels for each primitive. These labels are created using a canvas-based sprite system which always faces the camera for better readability.

Controls - Lana Glisic

In-game GUI

The GUI available while playing the game has been designed with several concerns in mind. Firstly, it is meant to take up as little space as possible in the user's field of vision, allowing the focus of the game to remain on the scene. The scene is populated with objects, and rotation around the central point of the scene is the user's main way of comparing these objects' visual qualities in order to arrive at the correct solution. This rotational area comprises a roughly circular space at the center of the screen, leaving the corners generally empty. For this reason, the GUI has been placed in a corner of the screen rather than its center. The top-left corner was chosen for visual immediacy, here assuming a left-to-right reading convention.

The GUI's color scheme has been designed to keep the user's attention on the objects in the scene and avoid detracting from the color differentiation task. As such, it has been designed to be mostly black with white text and outlines. This is intended to provide clear contrast and maximize readability while maintaining subtlety. By using a black and white color scheme, we also prevent the user from using hue comparisons with the GUI itself to their advantage.

The logo and circular design features of the GUI and its buttons are meant to evoke the color wheel, as well as common three-circle depictions of RGB color intersections.

In order to submit an answer and level up, the user first needs to type their answer into the blank text box centered within the top-left GUI, then either click "Submit" or press the Enter key. Since submission to the text box is necessary to progress the

game, and quick access leads to higher scores, several positions for the text box were considered. However, its position ultimately seemed best-suited to a part of the screen where the user wouldn't accidentally click when trying to rotate around the scene. The same reasoning underlies the choice of numbering objects instead of allowing an immediate object-clicking mechanism for submission. Rotation or zooming with the mouse could cause accidental selection of an unintended object, and potential visual obstruction if the selection were represented by shading or highlighting that object.

Lastly, the top-left GUI and menu screen contain a "How to Play" button that when clicked will overlay the screen with an explanation of the instructions. Additionally, the top-left GUI contains a "Regenerate" button that when clicked will change the position and color of objects in the scene while retaining the perceptual difficulty associated with the player's current level. This feature is meant to better accommodate individual variations in players' visual abilities along various parts of the color spectrum.

Game Mechanics - Amanda Sparks

The major components of the game mechanics are as listed: checking the appropriate answer, calculating the level changes, and making those changes into the level, and tracking the appropriate statistics. As the game progresses, it should be increasingly difficult to determine the correct answer. We determined that the best way to do this was to increase the number of possible 'distractions' that could confuse the player or draw their attention away from the differently colored object, as well as to decrease the offset of the different object. We also wanted to track the statistics for the player to display at the end. These statistics include their minimum time, their maximum time, how many levels they complete, a calculated score, and the minimum offset they were able to perceive.

To make the levels more difficult, there is an increasing number of objects, an increasing number of types of objects, an increasing number of textures, and a decreasing offset. This was done in 'sets'. Each type of change has a chosen number, and when the level increases by that chosen number. For example, the number of primitives starts at 3, and the chosen number is 3. Once the level reaches 3, the number of primitives increases by one to become 3. The formula for this is as follows:

$$value = [minimum\ value] + Math.floor(level / chosen\ number)$$

For the number of textures used and the number of primitive types, this value was then compared to the maximum number of each available, and the minimum value was used to prevent undefined behaviour.

This value was then used to create the appropriate number of primitives. One of these primitives is selected at random, and the color offset is applied to it. The appropriate primitive is logged into a variable within the state of the level. To check the level, the variable is compared to the input, and the appropriate actions are then taken. If the answer is correct, the statistics for the level are tracked through a dictionary.

To access the types of objects and the textures, there is an array of string variables for each one. This allows for the easy addition of more texture and further primitives.

Results

We play-tested our game, logging both our scores and statistics, including the total amount of time it took for us to finish the game. To allow for more impartial judgment, we also had an outside player test the game as well. Player T, as this outside player will be known, has requested not to be named. They experience the deuteranopia form of colorblindness, and expressed some issues with the more difficult levels of the game, but stated that it was easier to play than “I Love Hue”, the previous work referenced above.

Levels completed	Total time (minutes)	Score	Min. offset	Min. time (s)	Max. time (s)	Player
10	3	593	90	4.46	21.80	Amanda
31	11	1674	20	11.64	93.20	Lauren
29	6	741	20	29.98	119.78	Lauren
27	9	1070	30	4.54	126.84	Amanda
15	5	642	60	6.39	95.74	Player T
29	14	1263	30	5.21	131.45	Player T
27	3	1707	30	3.09	85.36	Lana
30	3	1060	20	4.26	152.02	Lana

Discussion

Ethical Concerns

Color Blindness

There are three major types of color blindness. The mildest is Anomalous Trichromacy, where a person has all three types of cones, but one is less sensitive. This often leads to difficulty distinguishing pale or muted colors, and in more severe cases, even vivid and pure colors can be confused. The next type is Dichromacy, which occurs when one type of cone is missing. People with Dichromacy can only perceive color wavelengths that are combinations of the remaining two cones, making it challenging to differentiate between saturated colors. The most severe form is Monochromacy, where only one type of cone is present, resulting in vision limited to varying shades of gray.

Our game uses a color algorithm that offsets each color channel by the same amount. This approach ensures that color differences affect only saturation, not hue, enabling individuals with color vision deficiencies to distinguish between different shades of the primitives. We designed the game not only to accommodate players with color vision deficiencies but also to potentially serve as a diagnostic tool for color blindness. Our goal is to create a more engaging and enjoyable alternative to traditional color blindness tests, while also raising awareness about color vision deficiencies. (ACM Ethical Principles 1.4)

Copyrights

We drew inspiration from the game *I Love Hue* and traditional color blindness tests; however, no elements from these sources were directly used in creating this game. The skybox was hand-drawn by a team member, and all textures were procedurally generated, ensuring no external image sources were used. This game was built using Edwin Webb's Three.js seed project, originally developed by Reilly Bova and later updated as a GitHub template by Adam Finkelstein and Joseph Lou. Proper attribution and credits are provided at the bottom of our GitHub repository. The game is currently hosted online at https://amanda-2.github.io/hueclue.github.io/. (ACM Ethical Principles 1.5)

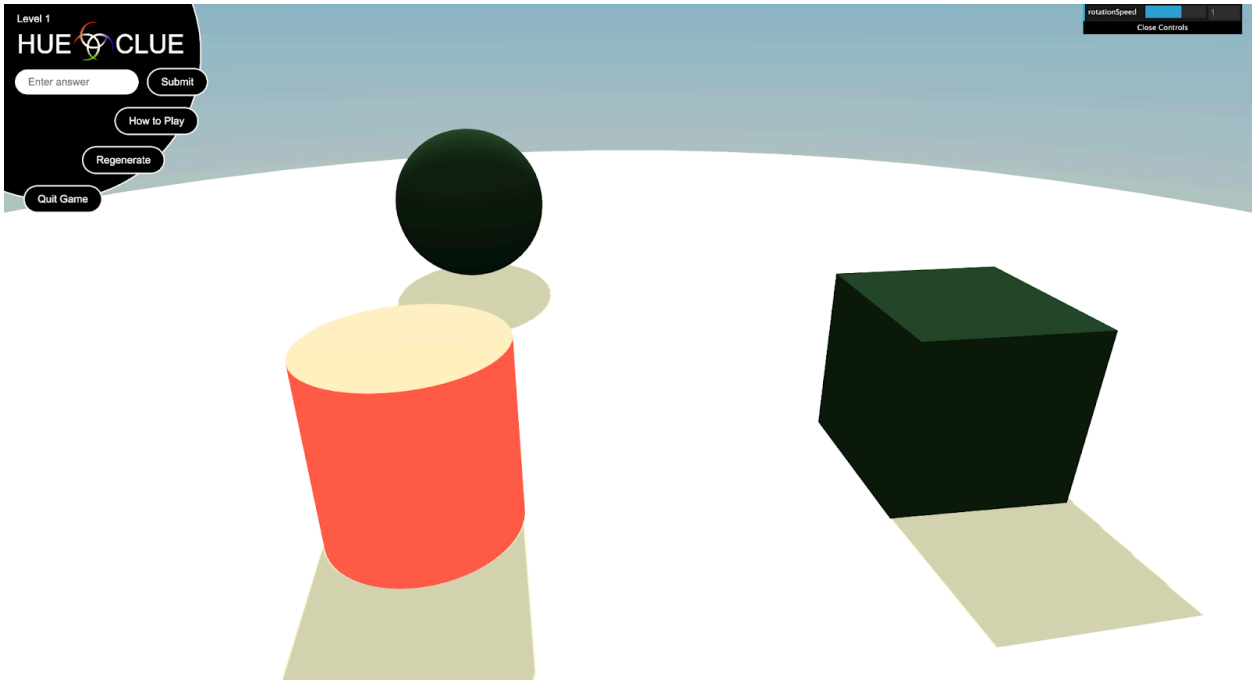
Future Work & Improvements

Accessibility Settings

For future work, we aim to implement accessibility settings in the start menu to accommodate various types of color blindness. These settings would allow individuals with color vision deficiencies to toggle optimized game modes. Each color blindness filter would adjust the game visuals to either enhance contrast or avoid problematic color hues. According to the Web Content Accessibility Guidelines (WCAG), accessible colors should have a minimum contrast ratio of 4.5:1. Since different types of color blindness have varying sensitivities to specific colors, we hope to address these differences through customizable accessibility settings.

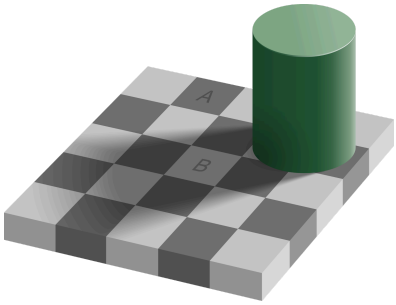
Clicking Interaction

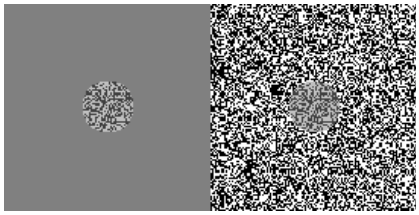
In addition to manually inputting the numbers of the primitives, we implemented a direct selection method through clicking. When the user hovers the cursor over a primitive, its emissive color turns red. Clicking on the primitive submits the selection for color difference verification. We believe this clicking approach provides a more intuitive user interaction compared to typing numbers manually. However, since the game focuses on color matching, overlaying a red emissive color impacts color perception and reduces accessibility for color vision deficient (CVD) users. For this reason, we opted to retain the number input method in our final implementation and leave the clicking feature as an option for future refinement.

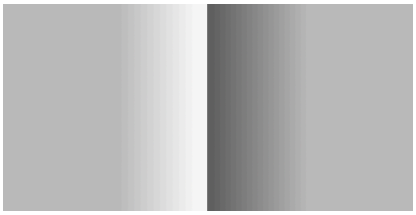


Lighting and Shadow Illusions

In the early stages of development, we explored using various lighting and shadow effects to introduce perceptual illusions and increase game difficulty. One famous example is the checker shadow illusion, where identical colors appear different when shadowed by other objects. We envision incorporating similar effects as a fun add-on to our game.







Additionally, the Chubb illusion, which occurs when color contrast varies significantly, offers an opportunity to extend the textural variances in our game.

Another example is the Cornsweet illusion, where gradient boundaries create the perception of color differences, akin to Mach bands. To build upon these concepts, we plan to implement gradient effects, along with texture and lighting variations, in more advanced levels of the game.

References

https://my.clevelandclinic.org/health/diseases/11604-color-blindness
https://en.wikipedia.org/wiki/Color_blindness
https://www.w3.org/WAI/standards-guidelines/wcag/
https://en.wikipedia.org/wiki/Checker_shadow_illusion
https://en.wikipedia.org/wiki/Chubb_illusion
https://en.wikipedia.org/wiki/Cornsweet_illusion
Zut Games, Ltd. (n.d.). *I love Hue (IOS, Android)*. Zut!
https://zutgames.com/i-love-hue.php

https://www.archimedes-lab.org/colorblindnesstest.html

Responsibilities of Each Group Member:

Amanda Sparks	Setting up starter code, deployment, original box code, game mechanics including level object code, checking answers, implementing variable level difficulty algorithms
Lauren Wang	Creating scene elements (primitive.js, textures.js, Floor.js, Skybox.js, BasicLights.js), main menu aesthetics, in-game menu aesthetics, carousel
Lana Glisic	Designed and coded GUI and initial menu screen; added GUI-related functionality including scene renewal, quitting, submitting an answer, and accessing instructions