Pixel-Art Scenes

Project Team Member 1 and Project Team Member 2
{xiaobin.wang001},{shen.xie001}@umb.edu
University of Massachusetts Boston

Figure 1: An example scene of 3D pixel art image.

ABSTRACT

In this project, we develop a system to automatically convert normal images into pixel art-style 3D scenes of cubes based on HTML5, Three.js and dui.js. It can load images, sample pixels, convert pixels to cube and render the 3D scenes. And it also provides a GUI to adjust the parameters and dynamically change the 3D scene in real time. The experimental results show that it perform well in different platforms.

KEYWORDS

3D pixel art, WebGL, Visualization, Three.js, dui.js

ACM Reference Format:
Project Team Member 1 and Project Team Member 2. 2022. Pixel-Art Scenes. In *CS460: Computer Graphics at UMass Boston, Fall 2022*. Boston, MA, USA, 4 pages. https://CS460.org

CS460, Fall 2022, Boston, MA

ACM ISBN 1337.
https://CS460.org

1 INTRODUCTION

Pixel art [6] is a form of digital art drawn with graphical software where images are built using pixels as the only building block. Pixel art started as a way for developers to create images using limited graphics and computing resources, and was not considered an art. But with the development of time, pixel art, as a special image style, has gradually been liked by people. There are many works to automatically convert normal images into pixel art style, but them can only generate 2D pixel art images.

Similar to 2d converting, we can transforming pixels into 3D cubes. In this project, we try to use technologies such as Three.js [1] and dui.js [3] to build an conversion system to generate pixel art-style 3D scenes from normal images automatically. In order to implement the system, we implemented functions such as image loading, pixel extraction, sampling, and scene rendering. Moreover, we also provide a useful GUI for easy setting of different parameters. Importantly, the 3D scene can be dynamically adjusted in real time according to changes in parameters.

2 RELATED WORK

There are already some great works on converting normal images into 2D pixel art style [2] [4]. But few 3D pixel art converting works were developed. We can try to transform pixels into 3D cubes in this project. Three.js [1] is an easy to use, lightweight, cross-browser,

general purpose 3D library used to create and display animated 3D computer graphics in a web browser using WebGL. And dui.js [3] is a extremely lightweight cross-browser graphic user interfaces library. In this project, we will use them to build a system to convert images into 3D pixel art style cubes.

3 METHOD

In this section, we will describe the design and implementations of the project in detail.

3.1 Implementation

In this project, we use **<canvas>** element of *HTML5* [5] to get the image pixel data, *dat.gui* for graphical user interface and parameters setting, and *three.js* load images and display 3D cubes. The main process of this project is as follows:

- Step 1: Basic scene settings: including background, camera and lights, etc.
- Step 2: Image loading: use *THREE.ImageLoader()* to load image.
- Step 3: Obtain image information: use the **convas** element to obtain the full resolution size and pixels information of the image. Figure 2 is an example code snippet for obtaining image information.

```
const loader = new THREE.ImageLoader();
loader.load(
  // resource URL
  imgUrl,

  // onLoad callback
  function (_image) {
    // use the image, e.g. draw part of it on a canvas
    _image.crossOrigin = "*";
    image = _image;
    cvs = document.createElement('canvas');
    cvs.width = image.width;
    cvs.height = image.height;

    ctx = cvs.getContext('2d');
    ctx.drawImage(image, 0, 0, cvs.width, cvs.height);

    const imageData = ctx.getImageData(0, 0, cvs.width, cvs.height);
    OimageData=imageData;

    generateBoxs(resizeImageData(imageData,sampleParams.width,sampleParams.height));
  },

  // onProgress callback currently not supported
  undefined,

  // onError callback
  function () {
    console.error('An error happened.');
  }
);
```

Figure 2: A code snippet for obtaining image information.

- Step 4: Sampling: the color pixels to be rendered is sampled from the original pixel matrix according to the size of the cube matrix. The code snippet for sampling is as Figure 3 shows.
- Step 5: Cube generation: generate the corresponding cube according to the color information of each pixel. The example code snippet for generation cubes is as shown in Figure 4.
- Step 6: Generate cube matrix: according to the coordinates of each pixel, add the cube to the corresponding position in 3D scene.

```
function resizeImageData(imageData, width, height) {
  const scale_w = width / image.width;
  const scale_h = height / image.height;
  const canvas = document.createElement('canvas');
  canvas.width = width;
  canvas.height = height;
  const ctx = canvas.getContext('2d');

  let scaledImageData = ctx.createImageData(width, height);

  for (var i = 0; i < height; i++) {
    for (var j = 0; j < width; j++) {
      const k = (i * width + j) * 4;
      const k2 = (Math.floor(i / scale_h) * image.width + Math.floor(j / scale_w)) * 4;
      scaledImageData.data[k] = imageData.data[k2];
      scaledImageData.data[k + 1] = imageData.data[k2 + 1];
      scaledImageData.data[k + 2] = imageData.data[k2 + 2];
      scaledImageData.data[k + 3] = imageData.data[k2 + 3];
    }
  }

  return scaledImageData;
}
```

Figure 3: A code snippet for Sampling.

```
function generateBoxs(imageData) {
  const geometry = new THREE.BoxGeometry(boxParams.width, boxParams.height, boxParams.depth);
  let x_offset = sampleParams.width * boxParams.gap / 2;
  let y_offset = sampleParams.height * boxParams.gap / 2;
  let z_offset = Math.max(x_offset, y_offset);
  groupCubes = new THREE.Group();
  for (var i = 0; i < sampleParams.height; i++) {
    for (var j = 0; j < sampleParams.width; j++) {
      const k = (i * sampleParams.width + j) * 4;
      const R = imageData.data[k];
      const G = imageData.data[k + 1];
      const B = imageData.data[k + 2];
      //const A = imageData.data[k + 3];
      const new_color = new THREE.Color(R / 255, G / 255, B / 255);
      const material = new THREE.MeshMatcapMaterial({ color: new_color });
      const cube = new THREE.Mesh(geometry, material);
      cube.position.set(boxParams.gap * j - x_offset, -boxParams.gap * i + y_offset, -z_offset);

      groupCubes.add(cube);
    }
  }
  scene.add(groupCubes);
};
```

Figure 4: A code snippet for Cube generation.

- Step 7. Render scene: render the generated scene for visualization.
- Step 8. GUI event: monitor the changes of parameters in the GUI, change the corresponding parameters, re-generate cubes and re-render the scene. Figure 5 is an example code snippet for parameters of cubes monitoring events setup.

3.2 Milestones

3.2.1 Milestone 1. We brainstormed different designs and discussed what features the project should achieve.

3.2.2 Milestone 2. For each feature, we investigated different possible implementations and finally decided on the specific methods to implement them. The main methods used are Three.js [1] and dui.js [3].

3.2.3 Milestone 3. We created basic scene and set up background, camera, lights and basic controls.

3.2.4 Milestone 4. Load image and obtain the size of the image and full pixels information. And achieve the function of pixel sampling of arbitrary width and height.

3.2.5 Milestone 5. Achieve the functions of converting pixels to cubes and rendering the cubes to 3D scene.

```
const boxFolder = gui.addFolder('Cubes')
boxFolder.add(boxParams, 'gap', 0, 100).step(1)
  .listen()
  .onChange((newValue) => {
    updateGap(boxParams.gap);
  });
boxFolder.add(boxParams, 'width', 0, 50).step(1)
  .listen()
  .onChange((newValue) => {
    updateBoxs(boxParams);
  });
boxFolder.add(boxParams, 'height', 0, 50).step(1)
  .listen()
  .onChange((newValue) => {
    updateBoxs(boxParams);
  });
boxFolder.add(boxParams, 'depth', 0, 50).step(1)
  .listen()
  .onChange((newValue) => {
    updateBoxs(boxParams);
  });
```

Figure 5: A code snippet for monitoring cubes parameters events.

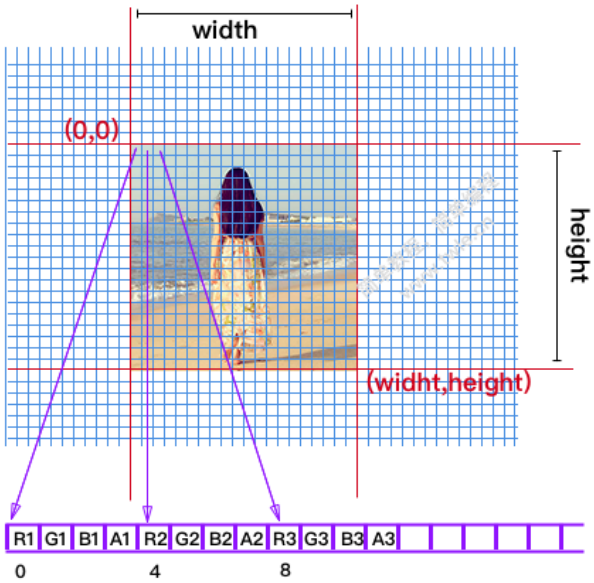

Figure 6: "RGBA" format of pixel

3.2.6 Milestone 6. Achieve the features of GUI and monitoring event to modify the cubes in real time.

3.2.7 Milestone 7. Performance optimization for better experience.

3.3 Challenges

There are some challenges we faced in the project.

- Challenge 1: Converting from pixels to cubes.
  Each pixel is represented by "RGBA" format, which is four one-byte values as Figure 6 shows. The ImageData.data interface for a <canvas> element represents the underlying pixel data in one-dimensional array containing the data in the RGBA order. And we need to map the pixel positions in one-dimensional array to cube positions in two dimensional array.
- Challenge 2: Sampling pixels with arbitrary width and height.
  We must store the origin image data for arbitrary width and height sampling. And if the GUI event notify the width or height of cube matrix was changed, we need to re-sampling and re-generate new cubes.
- Challenge 3: Callback functions for GUI events.
  There are different groups of parameters such as camera, cubes, and sampling rate, and each group has various parameters. So we need to implement each callback for each of these events.
- Challenge 4: Performance optimization.
  We also performed some optimization for better performance. For example, When width and height of cube matrix are large, there are a huge amount of cubes to render. We reuse the resource as much as possible to optimize the performance. And in some case of parameters change, we can just reuse the resource like geometries and colors instead of re-generating them.

4 RESULTS

In this section, the final results of this project are described to show some feature we implemented in project. And the origin image for testing is shown as Figure 7.

4.1 Performance statistics

As shown in Figure 8, a performance statistics module was add to monitor the performance of FPS and memory usage.

4.2 Camera parameters

As shown in Figure 9, different camera parameters were set.

4.3 Cube parameters

As shown in Figure 10, different cube parameters were set.

4.4 Sampling parameters

As shown in Figure 11, different sampling parameters were set.

4.5 Behaves on different devices

As shown in Figure 12, we test this project on different devices. All of them have normal behave. It is worth mentioning that the FPS on Android is 120 which is better than Windows(Chrome, 60 FPS) and MacOS(Safari, 60 FPS).

Figure 7: The origin image.

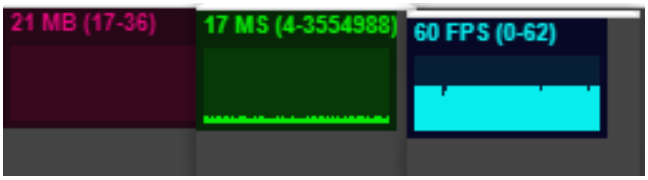

Figure 8: An example of performance statistics.

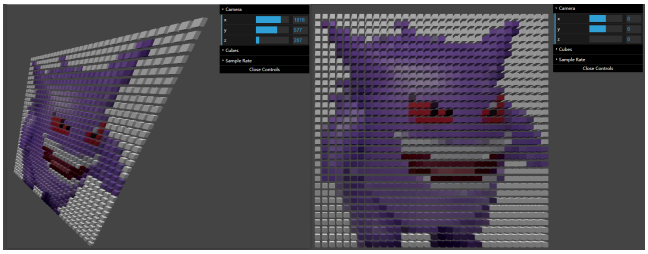

Figure 9: An example of different camera parameters.

5 CONCLUSIONS

Inspired by 2d pixel art conversion, we built a system to automatically generate pixel art-style 3D scenes from normal images by transforming pixels into 3D cubes. In order to implement the system, we implemented functions such as image loading, pixel extraction, sampling, and scene rendering based on some technologies such as HTML5, Three.js and dui.js. Moreover, we also provide a useful

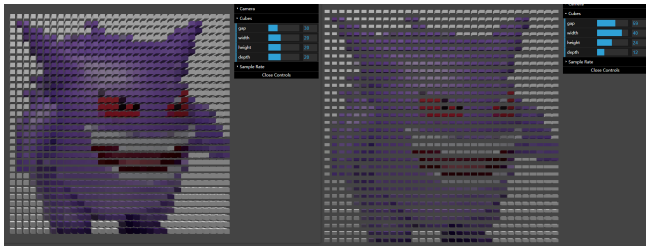

Figure 10: An example of different cube parameters.

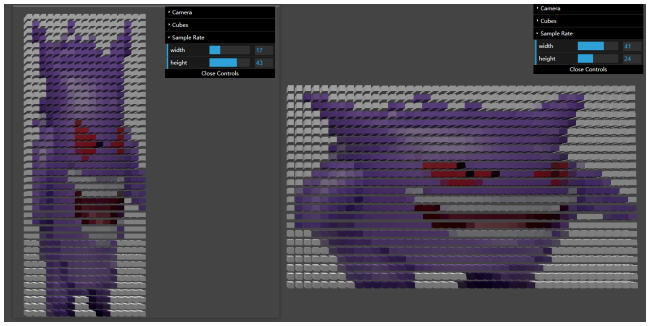

Figure 11: An example of different sampling parameters.

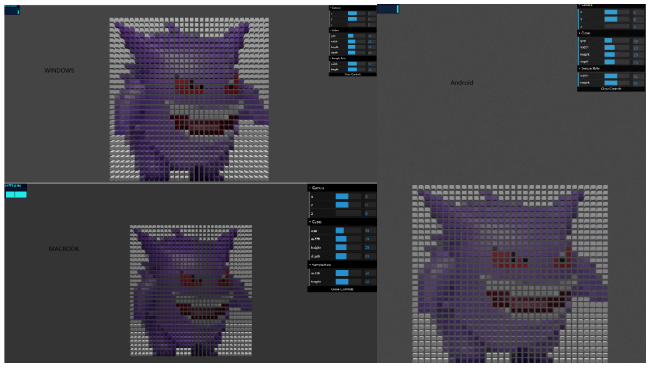

Figure 12: Behaves on different devices.

GUI for easy setting of different parameters And the 3D scene can be dynamically changed in real time according to the parameters. The experimental results show that it perform well in different platforms. This project may provide an example of a potential new way of creating content or games.

REFERENCES

[1] Ricardo Cabello et al. 2010. Three.js. *URL: https://github.com/mrdoob/three.js* (2010).
[2] colorinch. 2022. Cartoonize. *URL: https://www.cartoonize.net/pixelate-image* (2022).
[3] Dhrusya. 2018. dui.js. *URL: https://github.com/dhrusya/dui* (2018).
[4] giventofly. 2022. Pixel It. *URL: https://giventofly.github.io/pixelit/* (2022).
[5] mozilla. 2008. HTML5. *URL: https://developer.mozilla.org/en-US/docs/Glossary/HTML5* (2008).
[6] wikipedia. 2022. Pixel art. *URL: https://en.wikipedia.org/wiki/Pixel$_art$* (2022).