武汉大学计算机学院

本科生实验报告

**简单的类CAD软件**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：软件设计与体系结构

指 导 教 师 ：马于涛

小组成员 ：

张家瑜(组长) 2018302110024

李康龙 2018302110132

二○二○年十一月

**郑 重 声 明**

本人呈交的实验报告，是在老师的指导下，独立进行研究工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告的研究成果不包含他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

摘 要

本次软件体系架构设计与实现过程中，我们主要进行的是使用MarchingCube这一算法进行对体积数据的建模，并且自制了一个简单的笔刷工具用于涂抹修改模型。类似于使用Blender，ZBrush一类建模工具修改涂抹功能。

本次实验我们小组主要进行的是基于Unity引擎的开发，而Unity3D很方便地可以打包为WebGL包，用于在网页上显示运行结果，因此我们的侧重点不在Web服务上而是算法优化与实现上面。

本文档包括项目需求分析、主要设计细节介绍、测试流程以及团队成员的个人心得体会等等。

**关键词：MarchingCube、WebGL、Unity3D、体积建模。**

目录

[一.项目信息与团队分工 5](#_Toc55740717)

[1.1 项目代码库 5](#_Toc55740718)

[1.2 团队分工 5](#_Toc55740719)

[二.需求规格说明书 6](#_Toc55740720)

[2.1 引言 6](#_Toc55740721)

[2.1.1 编写目的 6](#_Toc55740722)

[2.2系统需求概述 6](#_Toc55740723)

[2.2.1用例图 6](#_Toc55740724)

[2.2.2 用例的概要描述 6](#_Toc55740725)

[2.2.3 假设和依赖 7](#_Toc55740726)

[2.3 功能需求 7](#_Toc55740727)

[2.3.1 绘制系统功能 7](#_Toc55740728)

[2.4 非功能需求 8](#_Toc55740729)

[2.4.1 易用性 8](#_Toc55740730)

[2.4.2 性能 9](#_Toc55740731)

[2.5. 运行环境规定 9](#_Toc55740732)

[三.系统设计说明书 10](#_Toc55740733)

[3.1 类图 10](#_Toc55740734)

[3.2 顺序图 11](#_Toc55740735)

[3.3 状态图 11](#_Toc55740736)

[3.4 设计模式 11](#_Toc55740737)

[四.项目实现 16](#_Toc55740738)

[4.1 算法知识储备 16](#_Toc55740739)

[4.1.1 MarchingCube 16](#_Toc55740740)

[4.1.2 部分Remarch以提高效率 20](#_Toc55740741)

[4.2 前端设计 21](#_Toc55740742)

[4.2.1 用户按钮 21](#_Toc55740743)

[4.2.2 拖动条控制笔刷属性 22](#_Toc55740744)

[4.2.3 用户视角控制 23](#_Toc55740745)

[4.3 后端功能实现 24](#_Toc55740746)

[4.3.1 ComputeShader简介 24](#_Toc55740747)

[4.4 web服务 25](#_Toc55740748)

**[1. 准备服务器](#_Toc55740749)** [25](#_Toc55740749)

**[2. 打包成WebGL](#_Toc55740750)** [25](#_Toc55740750)

[五.测试报告 27](#_Toc55740751)

[5.1 测试任务概述 27](#_Toc55740752)

[5.1.1 目标 27](#_Toc55740753)

[5.1.2 运行环境 27](#_Toc55740754)

[5.2 测试演示 27](#_Toc55740755)

[5.2.1 运行效果测试 28](#_Toc55740756)

[六.个人心得体会 28](#_Toc55740757)

**一.项目信息与团队分工**

本项目旨在开发一个与CAD类似的建模工具，而使用的主要算法是MarchingCube，顺带有实现Matcap的材质算法。前端直接使用的是Unity3D自带的UI模式，而后整个项目通过打包为WebGL（JavaScript）来实现web服务。

## 1.1 项目代码库

项目Github：<https://github.com/Monarchlee/SimpleCAD>

## 1.2 团队分工

张家瑜：负责主要的算法实现优化和文件类功能的实现，脚本的编写以及框架的配置。

李康龙：负责UI的实现和部分用户操控的交互实现，软件描述图的绘制以及文档的书写。

共同：软件的需求和设计。

# 二.需求规格说明书

## 2.1 引言

### 2.1.1 编写目的

本文档基于以下目的编写：

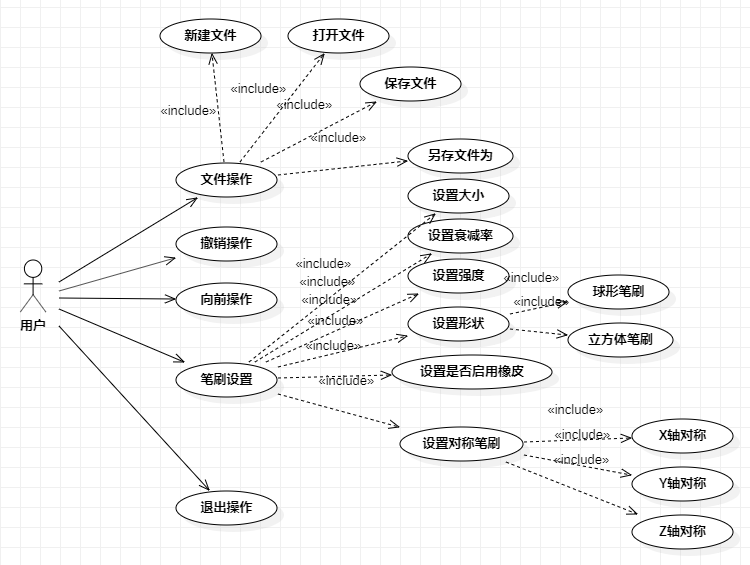
Ⅰ 使得对于此软件感兴趣的人能简单地看懂其功能

Ⅱ 使得原理一目了然

Ⅲ 存档以便将来的修改与优化

## 2.2系统需求概述

### 2.2.1用例图



### 2.2.2 用例的概要描述

| **主要参与者** | **优先级** | **用例名** | **用例概述** |
| --- | --- | --- | --- |
| **用户** | **高** | **文件操作** | **用户通过用例对体积模型文件操作** |
| **用户** | **高** | **撤销操作** | **用户通过撤销按钮撤销上一步的动作** |
| **用户** | **高** | **向前操作** | **用户通过向前按钮以向前进行一步** |
| **用户** | **高** | **笔刷设置** | **用户通过笔刷设置以设置不同的笔刷属性** |
| **用户** | **高** | **退出操作** | **用户通过退出按钮退出此软件** |

### 2.2.3 假设和依赖

1.用户必须具备基本的计算机知识。

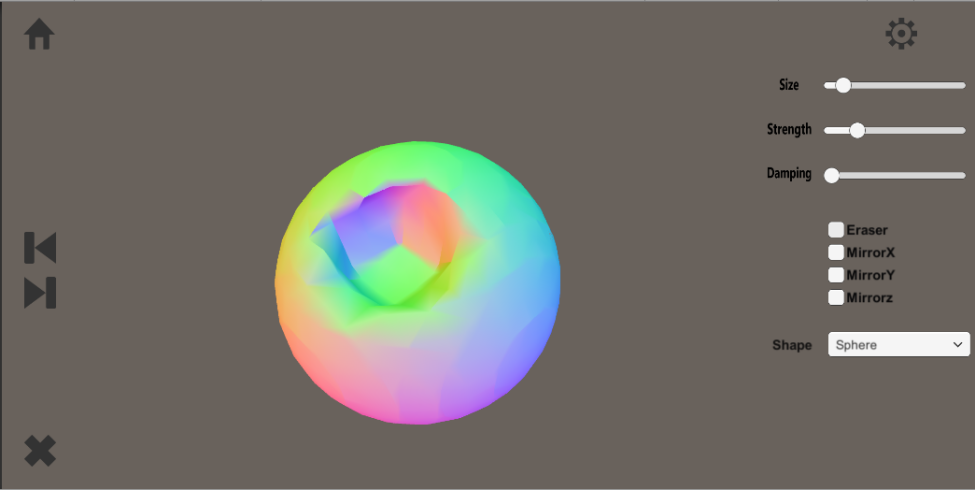
2.用户必须具备基本的建模知识。

## 2.3 功能需求

### 2.3.1 绘制系统功能

用户在软件中可以进行笔刷的调整以及文件的导入导出，主要功能是对应不同的笔刷进行对于体积模型的修改。

**页面如下:**

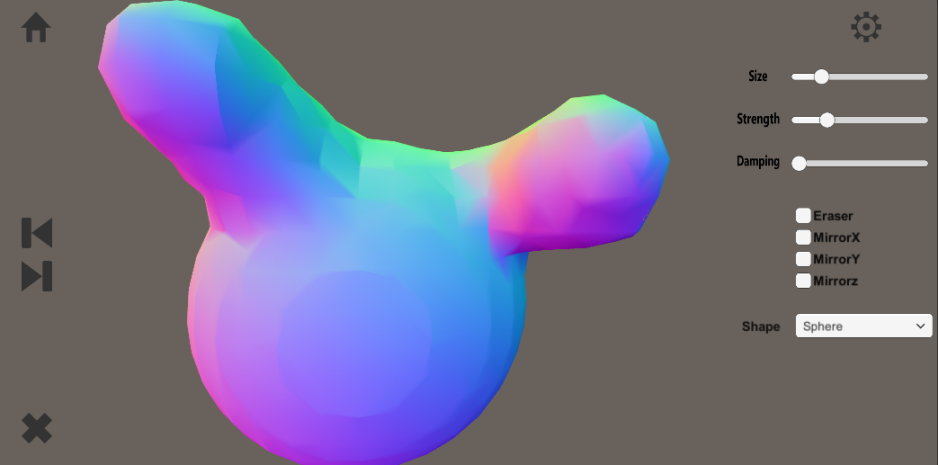


**操作流程如下:**

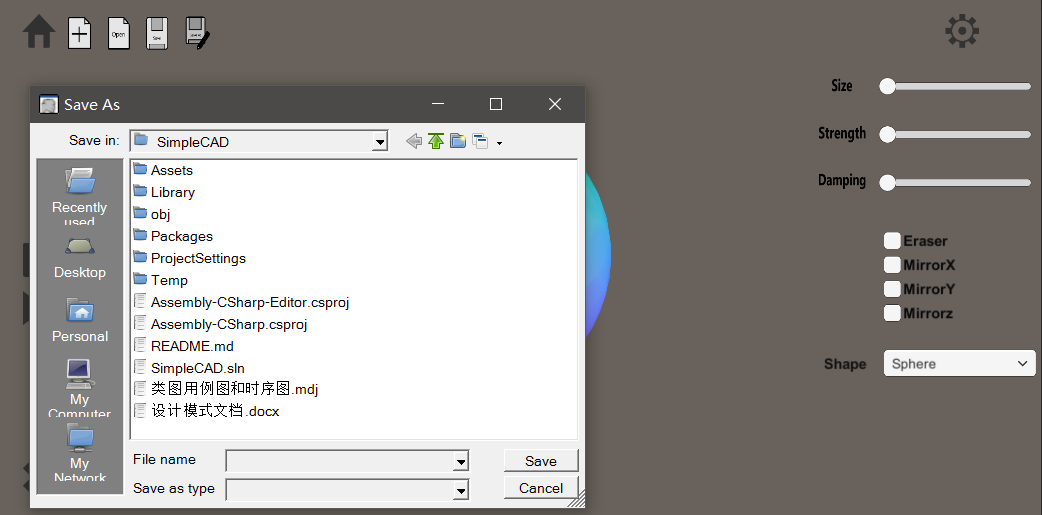
在右侧的几个状态栏中调整对应的笔刷的属性。

Size是笔刷大小，Strength是笔刷强度，Damping是衰减值，然后是启用橡皮与对称选项，最后的笔刷形状待实现，目前只有球形笔刷。

**操作界面如下：**



而后可以通过文件操作保存体积数据，演示如下：



## 2.4 非功能需求

### 2.4.1 易用性

| **需求分类** | **定义要素** | **需求内容** |
| --- | --- | --- |
| **用户界面** | **界面风格要求.** | **不冗余按钮或者UI，使得用户能很快找到对应功能按钮。** |
| **运行平台** | **运行环境要求** | **用户可以在支持unity3d的浏览器上面运行，也可以直接下载单机版进行本地操作。** |

### 2.4.2 性能

①达到在连接电源模式下流畅的效果

②在绘制的网格面较大的时候允许出现部分帧数低于10的情况

## 2.5. 运行环境规定

* 平台要求

系统必须在 CPU 为 Intel core i5 以上,内存为 4GB 以上, 操作系统为微软 win10以上。

* Web 浏览器

系统的用户界面必须在Chrome80.0 以上的浏览器正常显示.

# 三.系统设计说明书

## 3.1 类图

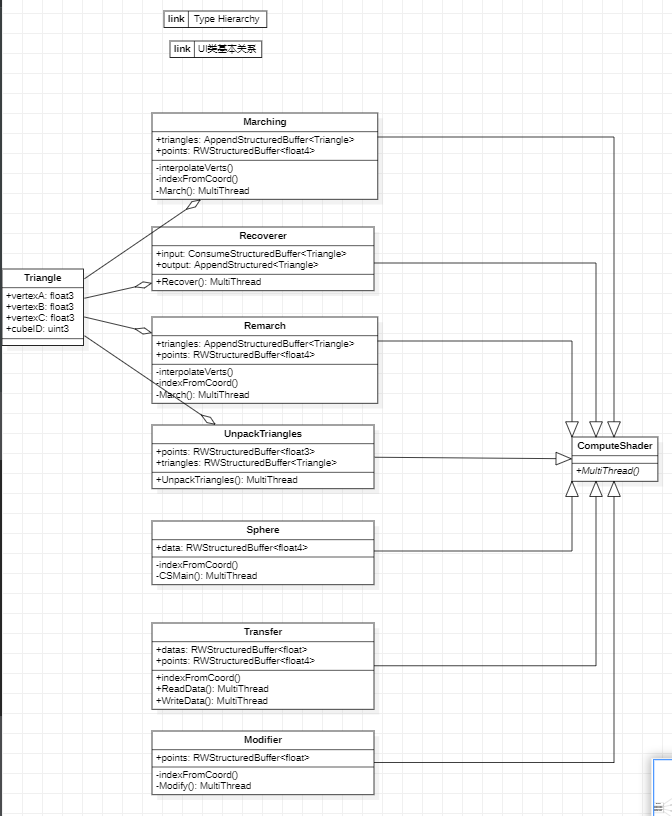


图1

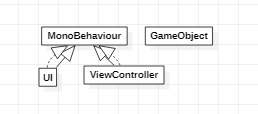
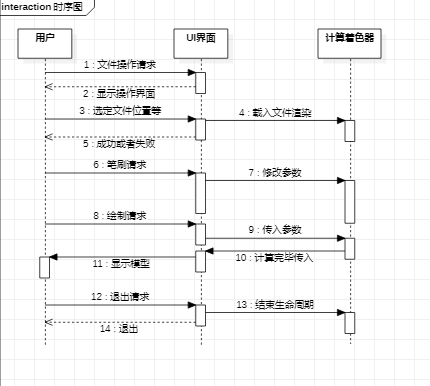


图2

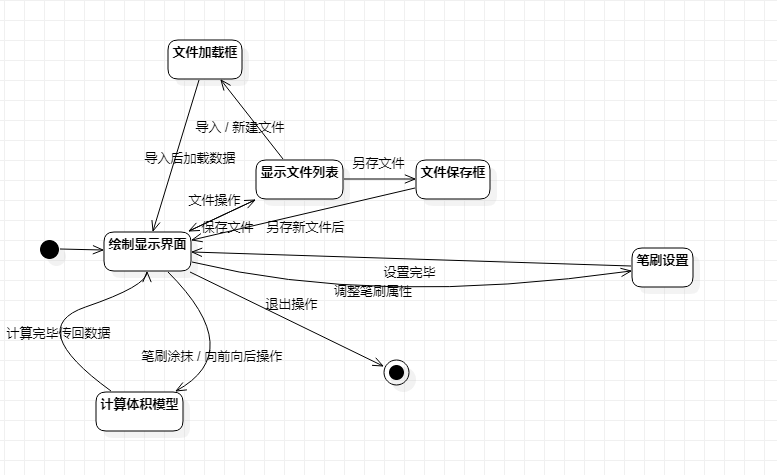
其中图1是主要的算法类所在的compute文件，都写在计算着色器里面，没有明显的类依赖关系。

图2是UI界面和视角控制的脚本类，都继承自Unity引擎的自带的脚本类。

## 3.2 顺序图



## 3.3 状态图



## 3.4 设计模式

本软件是类似CAD的建模软件的初步实现。借助Unity开发且主要使用marchingCube算法以绘制体积模型。

首先是使用到了C2风格的设计模式。由于和用户交互的操作十分频繁，所以图形界面是必不可少的。

1. **public** **class** Mouse : MonoBehaviour
2. {
3. Camera m\_Camera = **null**;
4. **public** Vector3 size = **new** Vector3(0.5f, 0.5f, 0.5f);
5. **private** Vector3 position = **new** Vector3(0, 0, 0);
6. [SerializeField] GameObject cursor = **null**;
7. [SerializeField] MeshGenerator generator = **null**;
9. [SerializeField] Vector3 canvasSize = Vector3.one \* 16;
10. [SerializeField] Vector3Int canvasDensity = Vector3Int.one \* 64;
11. [SerializeField] **float** initSphereRadius = 10;
13. [SerializeField] **float** radius = 1;
14. [SerializeField] **float** strength = 1;
15. [SerializeField] **float** damping = 0;
17. **public** **bool** mirror\_x = **false**;
18. **public** **bool** mirror\_y = **false**;
19. **public** **bool** mirror\_z = **false**;
21. // Start is called before the first frame update
22. **void** Start()
23. {
24. m\_Camera = Camera.main;
25. m\_Camera.depthTextureMode = DepthTextureMode.None;
27. cursor = Instantiate(cursor);
28. cursor.transform.localScale = size;
29. cursor.transform.position = position;
30. cursor.SetActive(**false**);
32. Volume sphere = **new** Volume(canvasSize, canvasDensity);
33. generator.InitVolume(sphere, initSphereRadius);
35. generator.InitBuffers(sphere);
36. generator.ReadData(sphere);
37. generator.MarchAll();
38. Mesh mesh = generator.GenerateMesh();
39. generator.SetMesh(mesh);
40. }
42. // Update is called once per frame
43. **void** Update()
44. {
45. Ray ray = m\_Camera.ScreenPointToRay(Input.mousePosition);
46. **if** (cursor.activeSelf == **false**)
47. cursor.SetActive(**true**);
48. **if** (Physics.Raycast(ray, **out** RaycastHit hit))//击中mesh
49. {
50. position = hit.point;
51. cursor.transform.position = position;
52. cursor.transform.LookAt(position + hit.normal);
53. **if** (Input.GetMouseButton(0))
54. {
55. Mirror(hit.point, **out** Vector3[] pos);
56. **foreach**(Vector3 p **in** pos) Modify(p, radius);
57. Mesh mesh = generator.GenerateMesh();
58. generator.SetMesh(mesh);
59. }
60. }
61. }

以上是Mouse.cs中对用户鼠标射线的检测代码以及部分处理函数。

其次，由于借助Unity引擎开发，不得不在其逻辑结构中进行脚本的绑定和函数调用。而其逻辑结构是顺序式的类似于层次的按时间顺序的。比如生命周期中的Start和Update。

因此其中有层次化的设计模式，见上述代码。

再者，在计算着色器即ComputeShader中需要GPU和CPU之间的数据传递，那么当然就需要一个双缓冲来匹配两者的速度。这类似于设计模式中的仓库模式。

1. // Each #kernel tells which function to compile; you can have many kernels
2. #pragma kernel Modify
4. **static** **const** **int** numThreads = 8;
6. float3 **base**;
7. float3 voxelCount;
8. float3 center;
9. **float** radius;
11. **float** strength;
12. **float** damping;
14. **int** numPointsX;
15. **int** numPointsY;
16. **int** numPointsZ;
18. RWStructuredBuffer<float4> points;
20. **int** indexFromCoord(**int** x, **int** y, **int** z) {
21. **return** y \* numPointsY \* numPointsX + x \* numPointsX + z;
22. }
24. [numthreads(numThreads,numThreads,numThreads)]
25. **void** Modify (int3 id : SV\_DispatchThreadID)
26. {
27. int3 bas = int3((**int**)**base**.x, (**int**)**base**.y, (**int**)**base**.z);
29. float3 cid = id + bas + float3(1,1,1);
30. **if** (cid.x >= numPointsX-1 || cid.y >= numPointsY-1 || cid.z >= numPointsZ-1 || cid.x <= 0 || cid.y <= 0 || cid.z <= 0) **return**;
31. **if**( id.x >= voxelCount.x-1 || id.y >= voxelCount.y-1 || id.z >= voxelCount.z-1) **return**;
33. **int** index = indexFromCoord(cid.x, cid.y, cid.z);
35. float3 pos = points[index].xyz;
36. **float** value = points[index].w;
38. //float dis = clamp(length(pos - center), 0, radius);
39. **float** dis = length(pos - center);
40. **float** weight = clamp(dis / radius,0,1);
42. **if**(dis > radius) **return**;
44. **float** delta = lerp(strength, strength \* (1 - damping), weight);
46. value = clamp(value + delta, -256, 256);
47. //value = value + delta;
49. points[index].w = value;
50. }

# 四.项目实现

## 4.1 算法知识储备

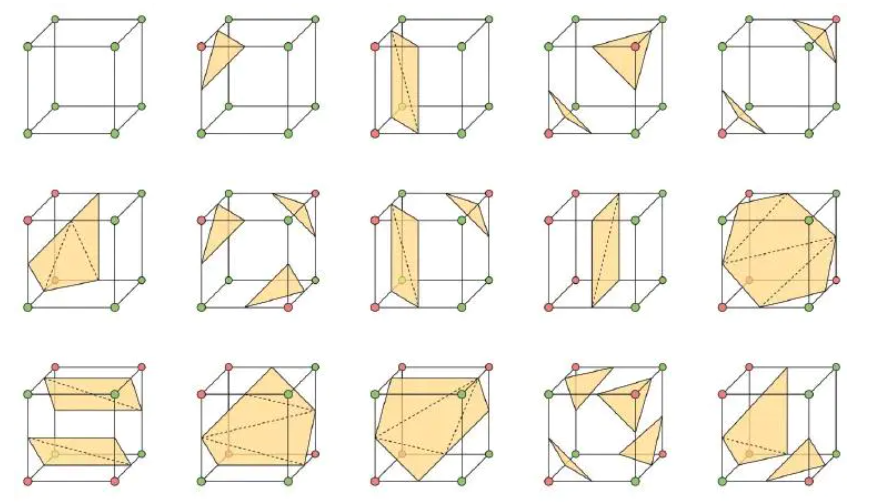
### 4.1.1 MarchingCube

MarchingCube算法主要用于根据点云数据确定空间中等值面的位置。我们通过在用户笔刷周边生成固定形状（笔刷形状）的点云以绘制对应的等值面来构成模型。

1. 将空间分为众多的六面体网格
2. 找到等值面经过的网格
3. 将多个等值面拓扑结构连接起来就得到了物体的表面

具体实现类似如下的过程：

首先根据每个六面体网格的顶点周围的点计算该点的平均场值，而后根据六面体八个顶点的不同场值使用线性插值的方法进行计算场值为0的等值面位置，具体有如下15种情况（图源网络）



具体根据六面体的对称性可得。

于是我们可以将这些数据存储下来以表格方式查询。

参见MarchTables文件中的数据。

这里不详细展示。

接下来的绘制过程在Marching文件中展示，

1. #pragma kernel March
2. #include "/Includes/MarchTables.compute"
4. **static** **const** **int** numThreads = 8;
6. **struct** Triangle {
7. float3 vertexC;
8. float3 vertexB;
9. float3 vertexA;
10. uint3 cubeID;
11. };
13. AppendStructuredBuffer<Triangle> triangles;
14. RWStructuredBuffer<float4> points;
16. float3 **base**;
18. **int** numPointsX;
19. **int** numPointsY;
20. **int** numPointsZ;
22. **float** isoLevel;
24. float3 interpolateVerts(float4 v1, float4 v2) {
25. **float** t = (isoLevel - v1.w) / (v2.w - v1.w);
26. **return** v1.xyz + t \* (v2.xyz-v1.xyz);
27. }
29. **int** indexFromCoord(**int** x, **int** y, **int** z) {
30. **return** y \* numPointsX \* numPointsZ + x \* numPointsZ + z;
31. }
33. [numthreads(numThreads,numThreads,numThreads)]
34. **void** March (int3 id : SV\_DispatchThreadID)
35. {
36. int3 bas = int3((**int**)**base**.x, (**int**)**base**.y, (**int**)**base**.z);
38. // Stop one point before the end because voxel includes neighbouring points
39. **if** (id.x >= numPointsX-1 || id.y >= numPointsY-1 || id.z >= numPointsZ-1) {
40. **return**;
41. }
43. id += bas;
45. // 8 corners of the current cube
46. float4 cubeCorners[8] = {
47. points[indexFromCoord(id.x, id.y, id.z)],
48. points[indexFromCoord(id.x + 1, id.y, id.z)],
49. points[indexFromCoord(id.x + 1, id.y, id.z + 1)],
50. points[indexFromCoord(id.x, id.y, id.z + 1)],
51. points[indexFromCoord(id.x, id.y + 1, id.z)],
52. points[indexFromCoord(id.x + 1, id.y + 1, id.z)],
53. points[indexFromCoord(id.x + 1, id.y + 1, id.z + 1)],
54. points[indexFromCoord(id.x, id.y + 1, id.z + 1)]
55. };
57. // Calculate unique index for each cube configuration.
58. // There are 256 possible values
59. // A value of 0 means cube is entirely inside surface; 255 entirely outside.
60. // The value is used to look up the edge table, which indicates which edges of the cube are cut by the isosurface.
61. **int** cubeIndex = 0;
62. **if** (cubeCorners[0].w < isoLevel) cubeIndex |= 1;
63. **if** (cubeCorners[1].w < isoLevel) cubeIndex |= 2;
64. **if** (cubeCorners[2].w < isoLevel) cubeIndex |= 4;
65. **if** (cubeCorners[3].w < isoLevel) cubeIndex |= 8;
66. **if** (cubeCorners[4].w < isoLevel) cubeIndex |= 16;
67. **if** (cubeCorners[5].w < isoLevel) cubeIndex |= 32;
68. **if** (cubeCorners[6].w < isoLevel) cubeIndex |= 64;
69. **if** (cubeCorners[7].w < isoLevel) cubeIndex |= 128;
71. // Create triangles for current cube configuration
72. **for** (**int** i = 0; triangulation[cubeIndex][i] != -1; i +=3) {
73. // Get indices of corner points A and B for each of the three edges
74. // of the cube that need to be joined to form the triangle.
75. **int** a0 = cornerIndexAFromEdge[triangulation[cubeIndex][i]];
76. **int** b0 = cornerIndexBFromEdge[triangulation[cubeIndex][i]];
78. **int** a1 = cornerIndexAFromEdge[triangulation[cubeIndex][i+1]];
79. **int** b1 = cornerIndexBFromEdge[triangulation[cubeIndex][i+1]];
81. **int** a2 = cornerIndexAFromEdge[triangulation[cubeIndex][i+2]];
82. **int** b2 = cornerIndexBFromEdge[triangulation[cubeIndex][i+2]];
84. Triangle tri;
85. tri.vertexA = interpolateVerts(cubeCorners[a0], cubeCorners[b0]);
86. tri.vertexB = interpolateVerts(cubeCorners[a1], cubeCorners[b1]);
87. tri.vertexC = interpolateVerts(cubeCorners[a2], cubeCorners[b2]);
88. tri.cubeID = id;
89. triangles.Append(tri);
90. }

93. }

首先是对于顶点的获取。

然后是对于这些顶点一一进行线性插值计算并根据结果选择对应的15种情况种的一种。所有这些计算过程都在并行的计算着色器中进行。

由于使用我们自定义的一个三角形结构体，其实就是三个顶点以及标志其所在六面体的ID。

### 4.1.2 部分Remarch以提高效率

在用户使用笔刷对体积模型进行修改的时候，简单想到但是都不会去使用的方法是进行完全的重新march以构建全新的体积模型。

于是我们采用了部分Remarch的方法来仅仅修改笔刷周围的点的场值同时只重新构建这部分受影响的六面体所决定的等值面。

具体代码见Modify.compute中的过程。

1. [numthreads(numThreads,numThreads,numThreads)]
2. **void** Modify (int3 id : SV\_DispatchThreadID)
3. {
4. int3 bas = int3((**int**)**base**.x, (**int**)**base**.y, (**int**)**base**.z);
6. float3 cid = id + bas + float3(1,1,1);
7. **if** (cid.x >= numPointsX-1 || cid.y >= numPointsY-1 || cid.z >= numPointsZ-1 || cid.x <= 0 || cid.y <= 0 || cid.z <= 0) **return**;
8. **if**( id.x >= voxelCount.x-1 || id.y >= voxelCount.y-1 || id.z >= voxelCount.z-1) **return**;
10. **int** index = indexFromCoord(cid.x, cid.y, cid.z);
12. float3 pos = points[index].xyz;
13. **float** value = points[index].w;
15. **float** dis = length(pos - center);
16. **float** weight = clamp(dis / radius,0,1);
18. **if**(dis > radius) **return**;
20. **float** delta = lerp(strength, strength \* (1 - damping), weight);
22. value = clamp(value + delta, -256, 256);
24. points[index].w = value;
25. }

## 4.2 前端设计

### 4.2.1 用户按钮

首先是对基本的用户按钮的设计与实现，直接使用Unity中的UI控件可以很简单的实现，下面是文件按钮弹出内置其它按钮的例子。

1. **public** **void** FileClick()
2. {
3. isDisplay = !isDisplay;
4. rootFile.transform.Find("New").gameObject.SetActive(isDisplay);
5. rootFile.transform.Find("Open").gameObject.SetActive(isDisplay);
6. rootFile.transform.Find("Save").gameObject.SetActive(isDisplay);
7. rootFile.transform.Find("SaveAs").gameObject.SetActive(isDisplay);
9. }

### 4.2.2 拖动条控制笔刷属性

使用Unity中的scrollbar控件以调整笔刷的强度，大小以及衰减值。其中这三个值都按比例调控，通过调整最大值可以增加上限。

1. **public** **void** SizeChange()
2. {
3. Slider slider = rootBrush.transform.Find("BrushSize").GetComponent<Slider>();
4. Mouse.radius = slider.value;
5. }
6. **public** **void** StrengthChange()
7. {
8. Slider slider = rootBrush.transform.Find("BrushStrength").GetComponent<Slider>();
9. Mouse.strength = Mouse.iseraser? -slider.value:slider.value;
10. }
11. **public** **void** DampingChange()
12. {
13. Slider slider = rootBrush.transform.Find("BrushDamping").GetComponent<Slider>();
14. Mouse.damping = slider.value;
15. }

而橡皮的实现只需要将strength改成相反数即可。

### 4.2.3 用户视角控制

鼠标中键控制平移，滚轮控制缩放，左键涂抹。

1. **public** **class** ViewController : MonoBehaviour
2. {
3. **public** **float** rotateSpeed = 3;
4. **public** **float** moveSpeed = 2;
5. **public** **float** scrollSpeed = 3;
6. **public** **float** dragSpeed = 10;
8. **void** FixedUpdate()
9. {
10. Vector2 mouse = **new** Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
11. Vector3 movement = **new** Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical") + Input.GetAxis("Mouse ScrollWheel") \* scrollSpeed) \* moveSpeed;
13. **if**(Input.GetMouseButton(2))
14. {
15. Vector2 drag = mouse \* dragSpeed;
16. movement -= (Vector3.up \* drag.y + Vector3.right \* drag.x);
17. }
19. **if**(Input.GetMouseButton(1))
20. {
21. Vector2 rotate = mouse \* rotateSpeed;
22. transform.Rotate(Vector3.up, rotate.x, Space.World);
23. transform.Rotate(transform.right, -rotate.y, Space.World);
24. }
26. transform.Translate(movement, Space.Self);
27. }
28. }

左键涂抹主要使用自带的光线追踪来实现。

## 4.3 后端功能实现

主要是使用Unity中的计算着色器自带的并行功能实现，并在GPU中进行主要运算和渲染。

### 4.3.1 ComputeShader简介

Compute Shader是一种技术，是微软DirectX 11 API新加入的特性，在Compute Shader的帮助下，程序员可直接将GPU作为并行处理器加以利用，GPU将不仅具有3D渲染能力，也具有其他的运算能力，也就是我们说的GPGPU的概念和物理加速运算。

对于Unity中的compute shader需要脚本的驱动。

C# Script 创建渲染所需的信息包括模型、贴图、参数等然后调用Compute Shader

Compute Shader计算出新的模型位置 对贴图进行处理 改变参数等

Vertex Fragment Shader获得Compute Shader产生的信息进行渲染

主要的格式需要定义缓冲以及线程组和线程数。这里不再赘述。

## 4.4 web服务

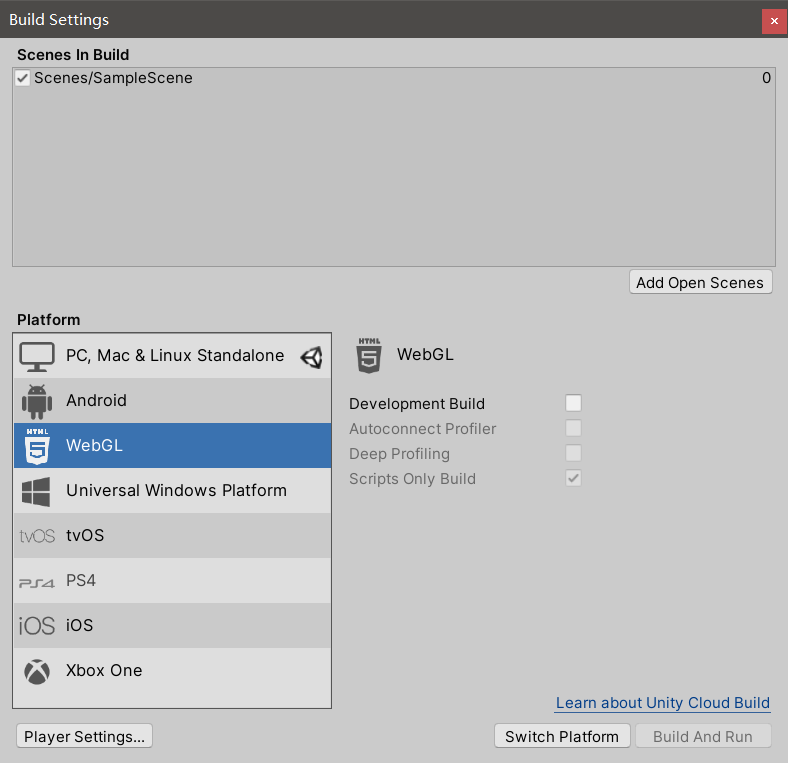
本章节介绍如何实现unity3d打包成webgl并访问。

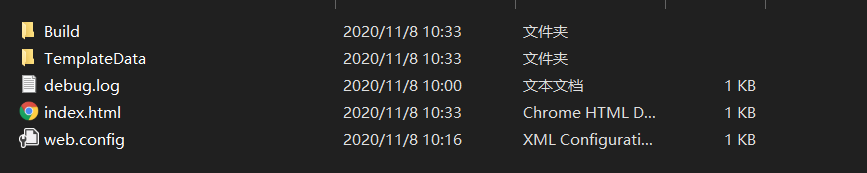
**1. 准备服务器**

这里直接使用本地IIS网站访问作为示例，实际过程中可以购买阿里云服务器等进行部署。

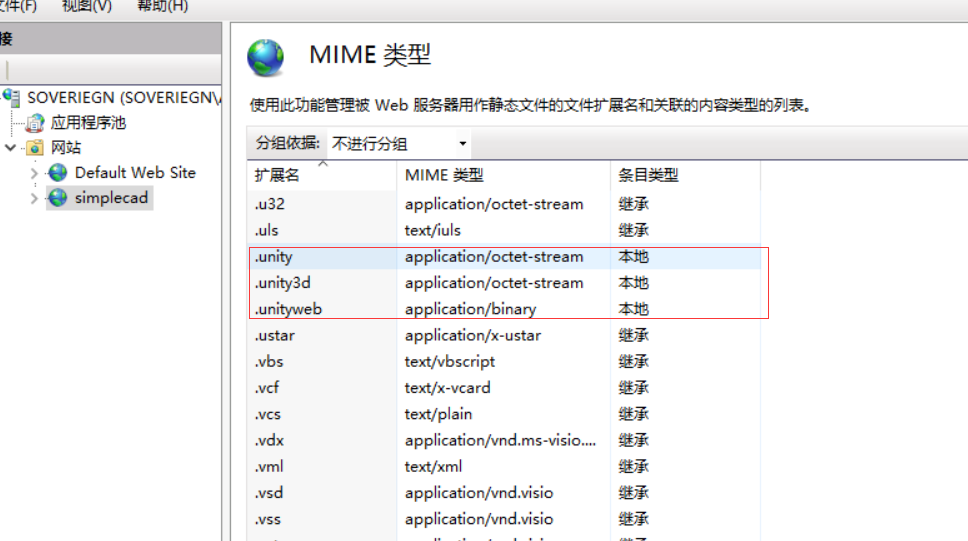
**2. 打包成WebGL**

直接使用Unity3D的插件（在设置中安装即可）来打包成WebGL。

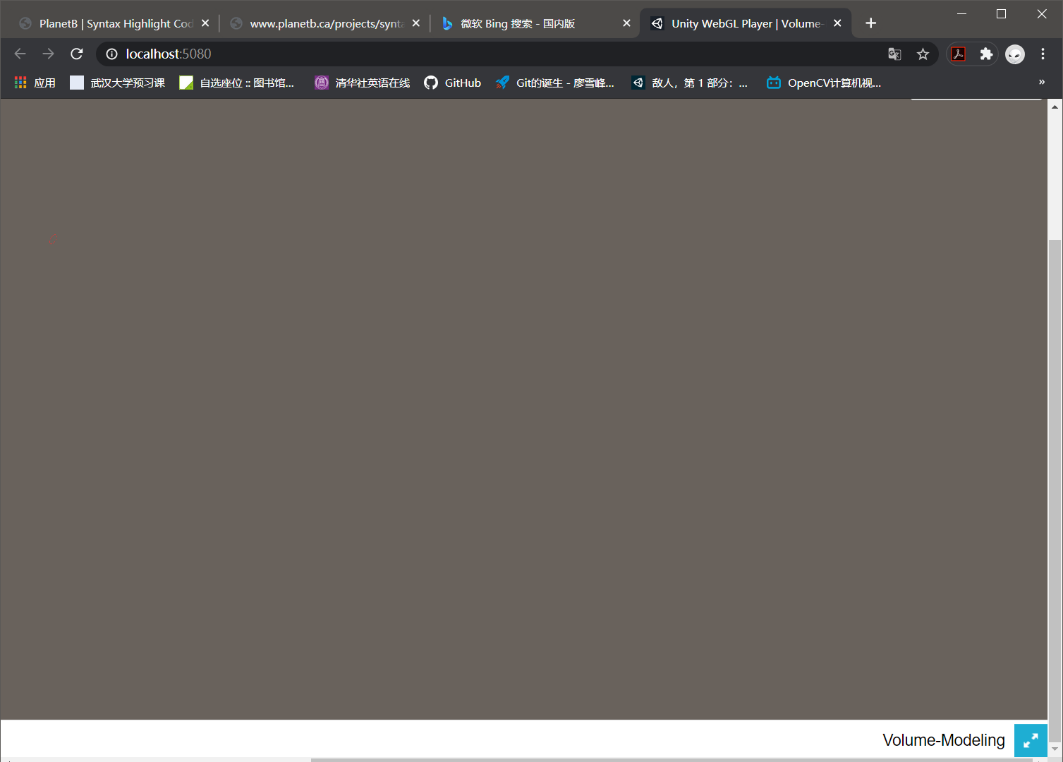
如图所示

打包后的结果如下：

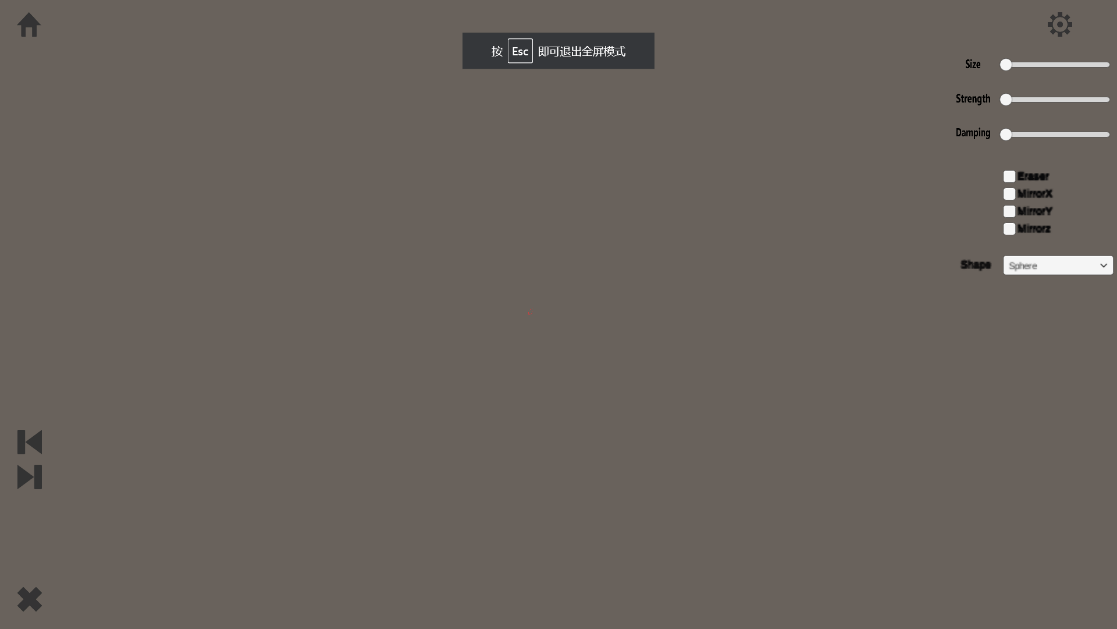
而后在本地部署IIS网站。

部署完毕后如图：

注意其中需要添加mime类型如图。

然后就可以通过浏览器访问，目前部署在本地。

点击全屏可进入，效果图如下：



# 五.测试报告

## 5.1 测试任务概述

### 5.1.1 目标

主要测试是否能流畅地实现涂抹以及消除等基本功能。

### 5.1.2 运行环境

* 平台要求

系统必须在 CPU 为 Intel core i5 以上,内存为 4GB 以上, 操作系统为微软 win10以上。

* Web 浏览器

系统的用户界面必须在Chrome80.0 以上的浏览器正常显示.

## 5.2 测试演示

### 5.2.1 运行效果测试

# 六.个人心得体会

·**张家瑜**

本次实验是对一直以来学习的图形算法的实践，切实的感觉到理论和实践的差距。在编码过程中犯下的错误包括没有将三角形打包压栈、没有考虑Mesh的顶点数量上限和写错了顶点的寻址函数。在算法的实现过程中，极大的提升了自己Debug的能力。除此之外，在Generator的搭建上下了很多功夫，对这个核心类的设计有了不小的感悟。看到一个真实的软件从完全无法运行到符合预期是一件令人欣喜的事情。

·**李康龙**

本次实验，跟着大佬学习了很多关于unity中尤其是compute shader的知识，也详细了解了MC算法的原理以及部分matcap。同时从零开始的软件架构也帮助我们逐渐规划好这个软件的走向以及实现方式。尤其是初期对软件需求文档的书写帮助我们理清楚了软件的具体模块功能的划分。同时对多个设计模式的选取以及参看相似类型软件的一般设计模式帮助我们更好地规划了软件将来的扩展方向以及实现过程的细节。

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | **角色** | **贡献分** | **可验证贡献** |
| 张家瑜 | 组长  后端 | 60 | 主要代码编写者，工程文件中的计算着色器的编写，marchingcube的实现，matcap的实现。用户视角的优化。 |
| 李康龙 | 前端 | 40 | UI的编写者，软件描述的图例绘制者。实验报告的整理，web服务的迁移。 |

※可见GitHub提交记录。

**教师评语评分**

评语：

评分：

评阅人：

2020 年 月 日