Final Report For MineCube

项目地址：MineCube on Github

notice：直接阅读Markdown体验更好

Final Report For MineCube

- Members
- MineCube简介
- 开发环境
- 第三方库
- 实现功能
  - Basic
  - Bonus
- 实现功能点简介
  - Camera Roaming
  - Simple lighting and shading(blinn-phong)
  - Texture mapping
  - Shadow mapping
  - Model import/export
  - Sky Box
  - Display Text
  - Complex Lighting （复杂光照: Gamma矫正）
  - Cloth Simulation
  - 3D拾取
  - 帧缓存特效
- 遇到的问题和解决方案
- 小组成员分工

Members

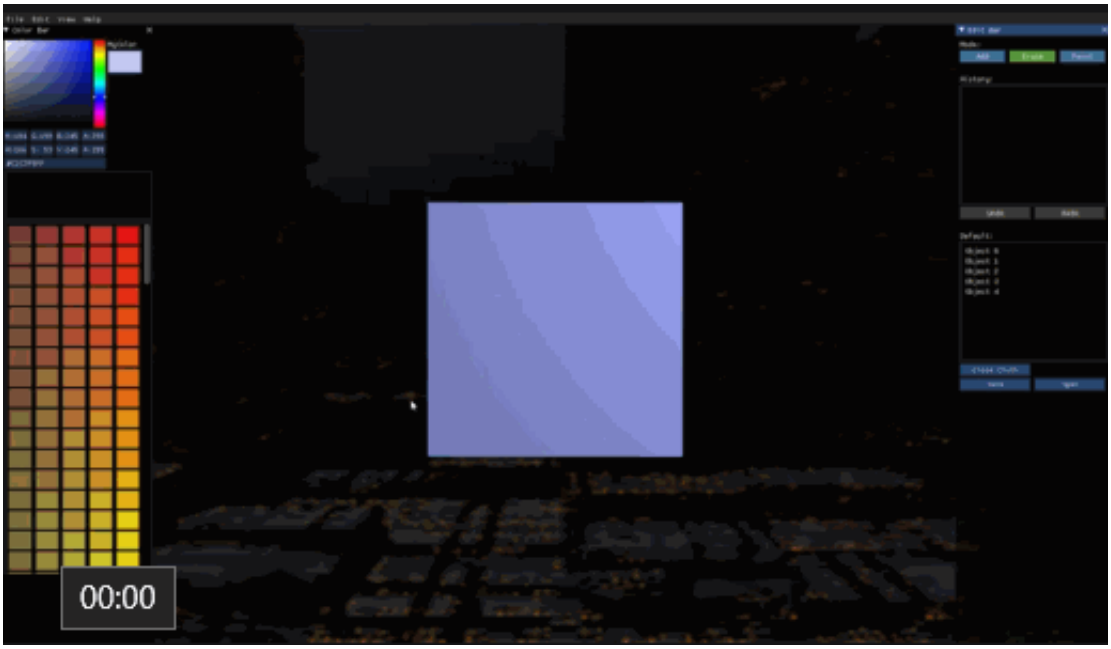
学号	姓名	Github
15331229	罗剑杰	Johnny Law
15331310	吴博文	Bob Wu
15331304	王治鋆	Jarvis
15331260	邱兆丰	mgsweet
15331335	徐海洋	Hiyoung.Tsui

MineCube简介

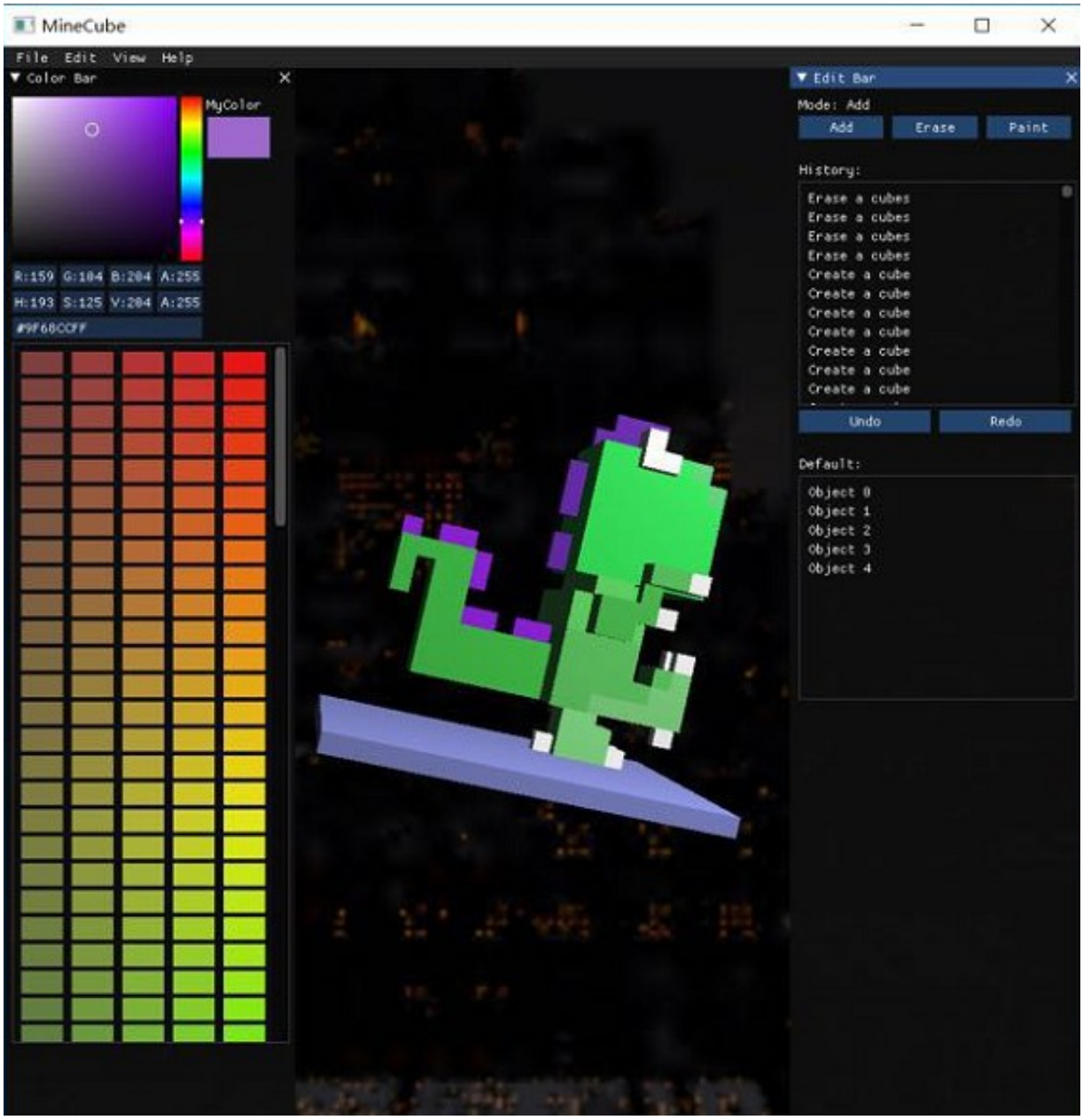
> **A sample voxel editor based on OpenGL 3.3+**, inspired by MagicaVoxel.
>
> Support Windows 10 currently.
>
> A final project originally of 5 undergraduate students for the course Computer Graphics, SYSU.
>
> **A good OpenGL learning example** for the green hand in Computer Graphics, while maybe not the best practice in the related field.

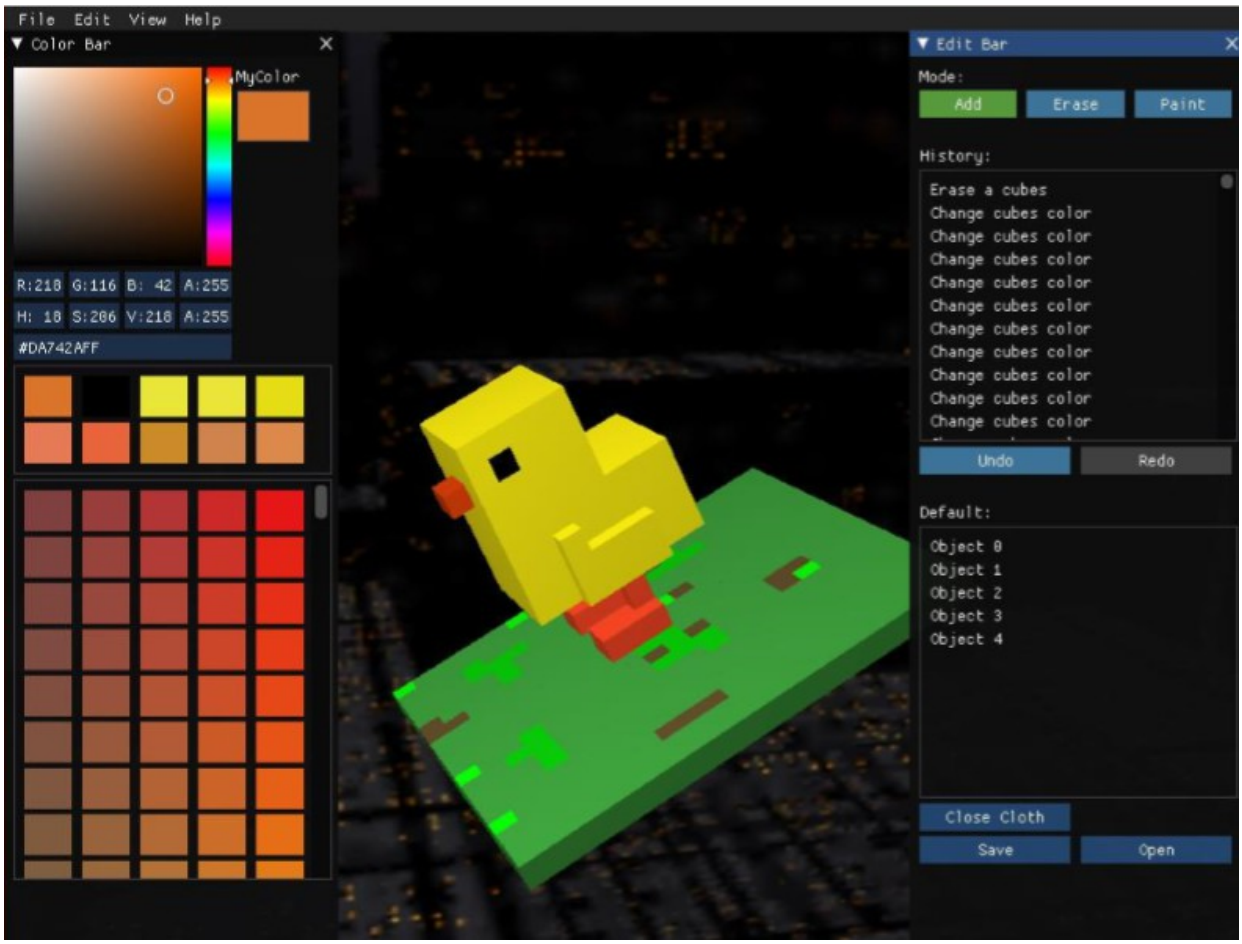
MineCube是一款受到MagicaVoxel启发的而开发的体素编辑器，通过操作小方块创造任何你的所想！

我们的成果！



上面为 gif 动图，可以到 `doc/` 下观看具体效果。


MineCube
File Edit View Help
Color Bar
MyColor
R:159 G:104 B:204 A:255
H:193 S:125 V:204 A:255
#9F68CCFF
Edit Bar
Mode: Add
Add
Erase
Paint
History:
Erase a cubes
Erase a cubes
Erase a cubes
Erase a cubes
Create a cube
Create a cube
Create a cube
Create a cube
Create a cube
Create a cube
Create a cube
Undo
Redo
Default:
Object 0
Object 1
Object 2
Object 3
Object 4



点击程序右下角的 `default` 模型选项查看我们已经画好的创意！

开发环境

Windows + OpenGL 3.3+ +Visual Studio 2015

第三方库

- GLAD
- GLFW Master branch
- GLM 0.9.8.5
- imgui v1.60
- nlohmann::json v3.1.2

具体部署流程

实现功能

Basic

- Camera Roaming
- Simple lighting and shading(blinn phong)
- Texture mapping
- Shadow mapping
- Model import/export

Bonus

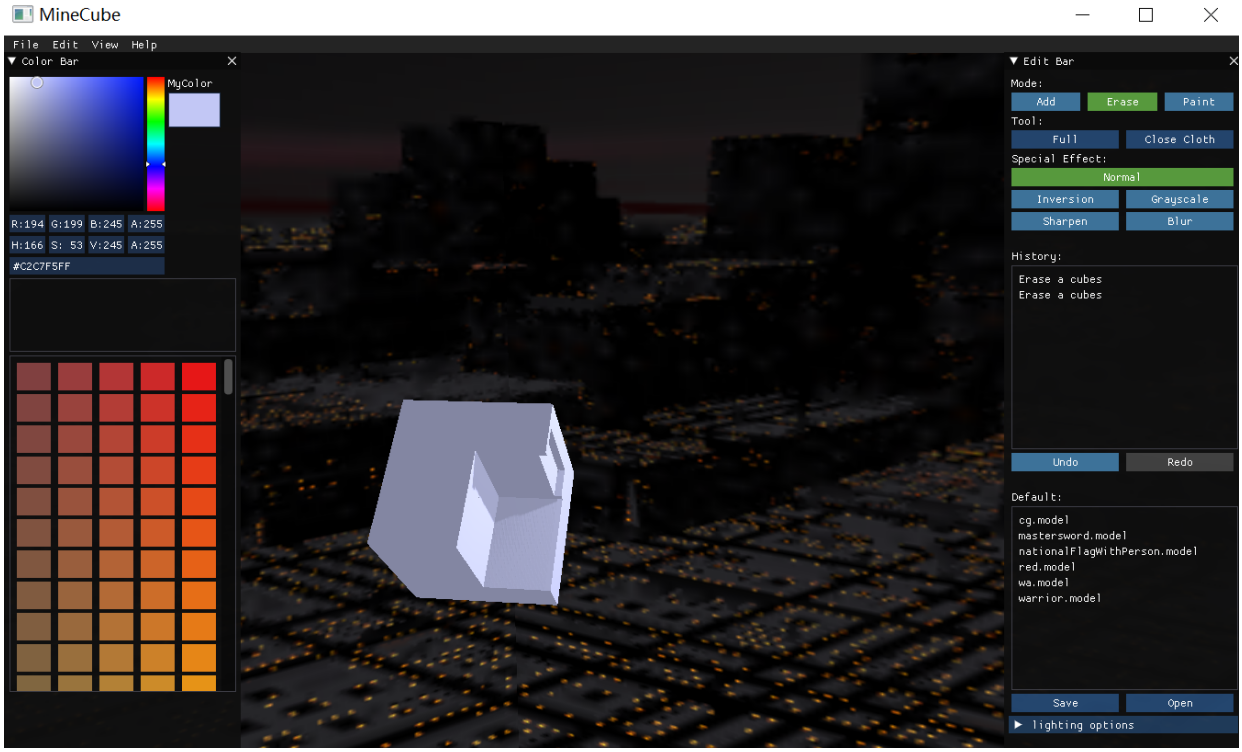
- Sky Box (天空盒)
- Display Text（显示文字）
- Complex Lighting （复杂光照: Gamma矫正）
- Cloth Simulation（织物模拟）
- Gravity System (织物模拟的重力系统)
- Particle System（织物模拟的粒子系统）
- 3D拾取
- 帧缓存特效

实现功能点简介

> Basic只介绍该点在项目中的应用，Bonus要介绍具体实现原理
>
> 每个点都要附上结果截图

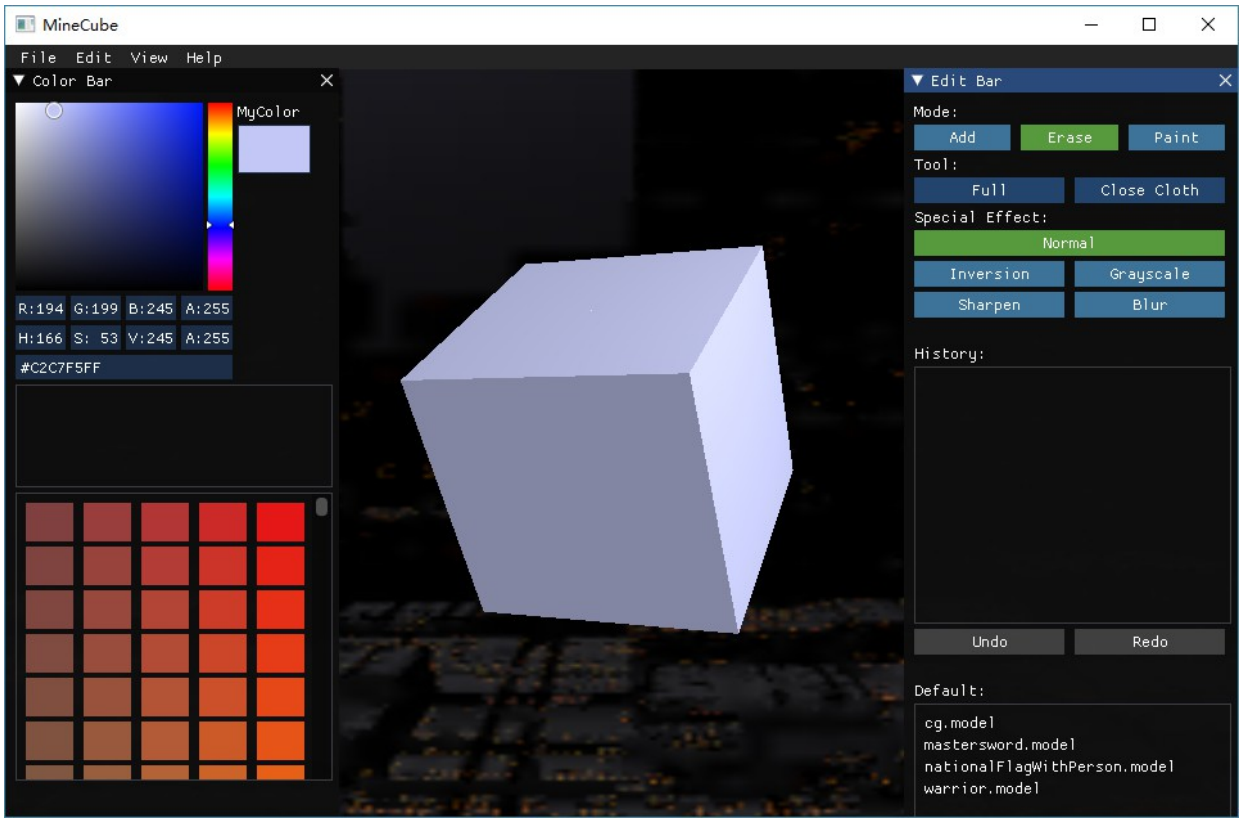
Camera Roaming

按`v`键可以切换FPS模式，进行自由移动和观察。使用`WASD`移动位置，鼠标移动来控制观察方向。



Simple lighting and shading(blinn-phong)

在场景的右上方有一个光源，对整个物体实现 blinn-phong 的光照效果，使得层次结构更加真实。

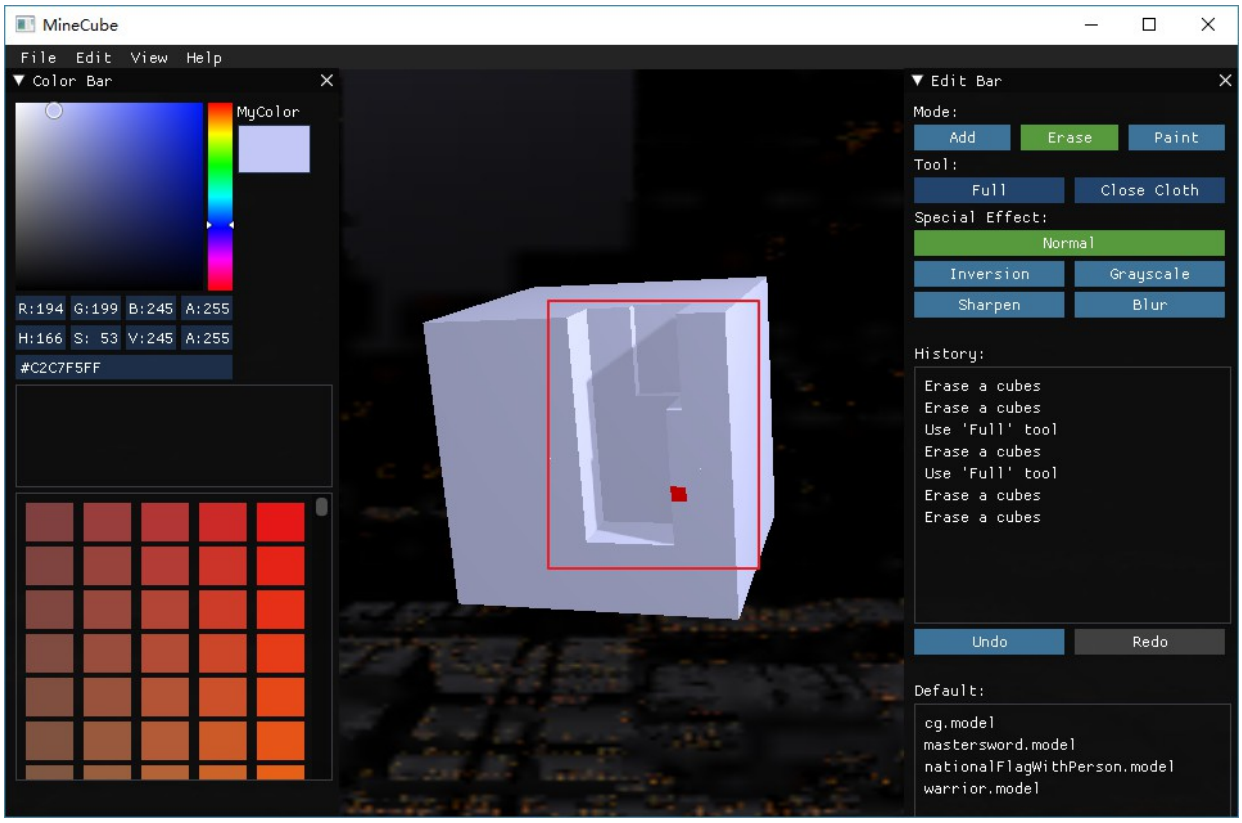


Texture mapping

文本渲染和天空盒的实现中运用了纹理映射的知识。

Shadow mapping

当部分小立方体被挖去之后，在适当的位置出现阴影，使得层次结构更加真实。



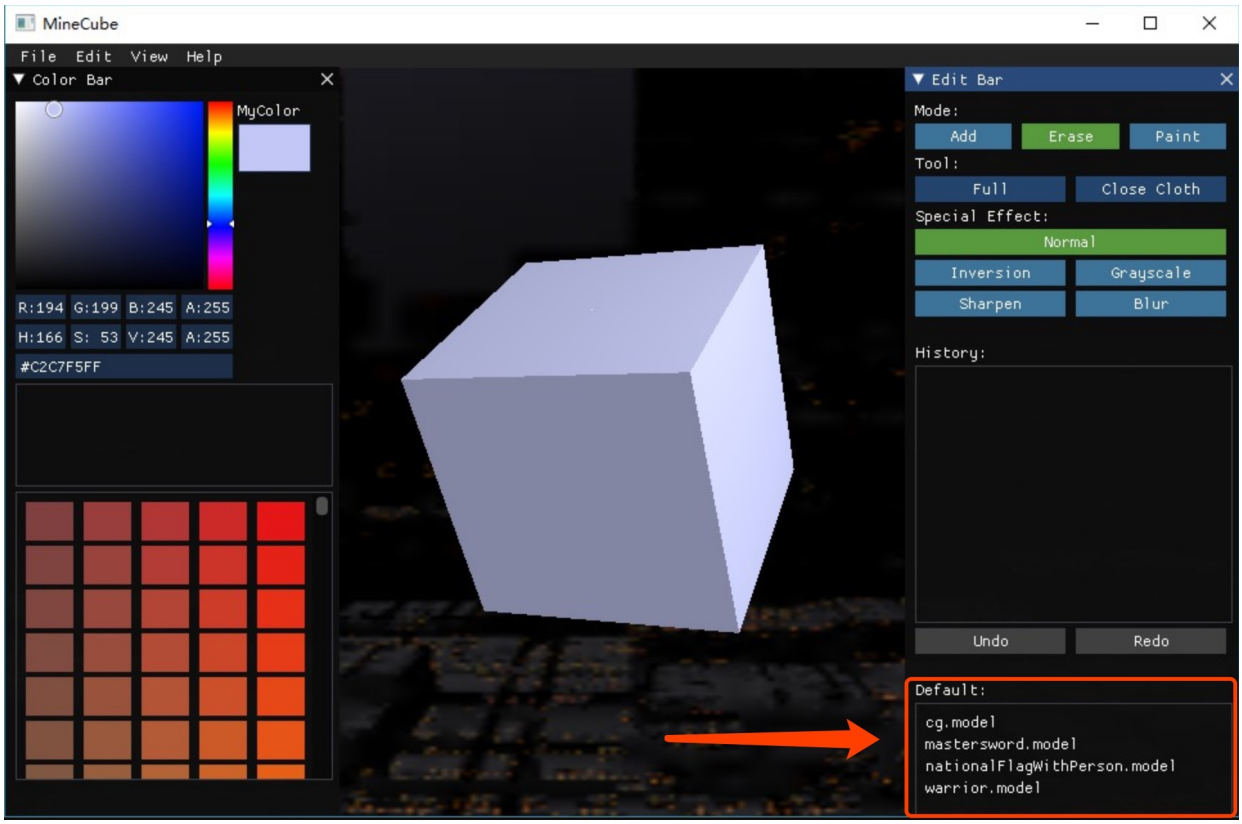
Model import/export

可以将创作的结果保存/载入，这里我们没有利用传统的`obj`格式模型，而是将模型格式定义为JSON

通过调用`nlohmann::json`生成JSON字符串，随后写入文件，完成模型导出

读入模型文件，使用`nlohmann::json`解析JSON字符串，将其中的信息恢复成CPP Object即完成了模型导入的过程。

在`Default`中可以导入我们预设的模型，模型保存在`Asset`目录下

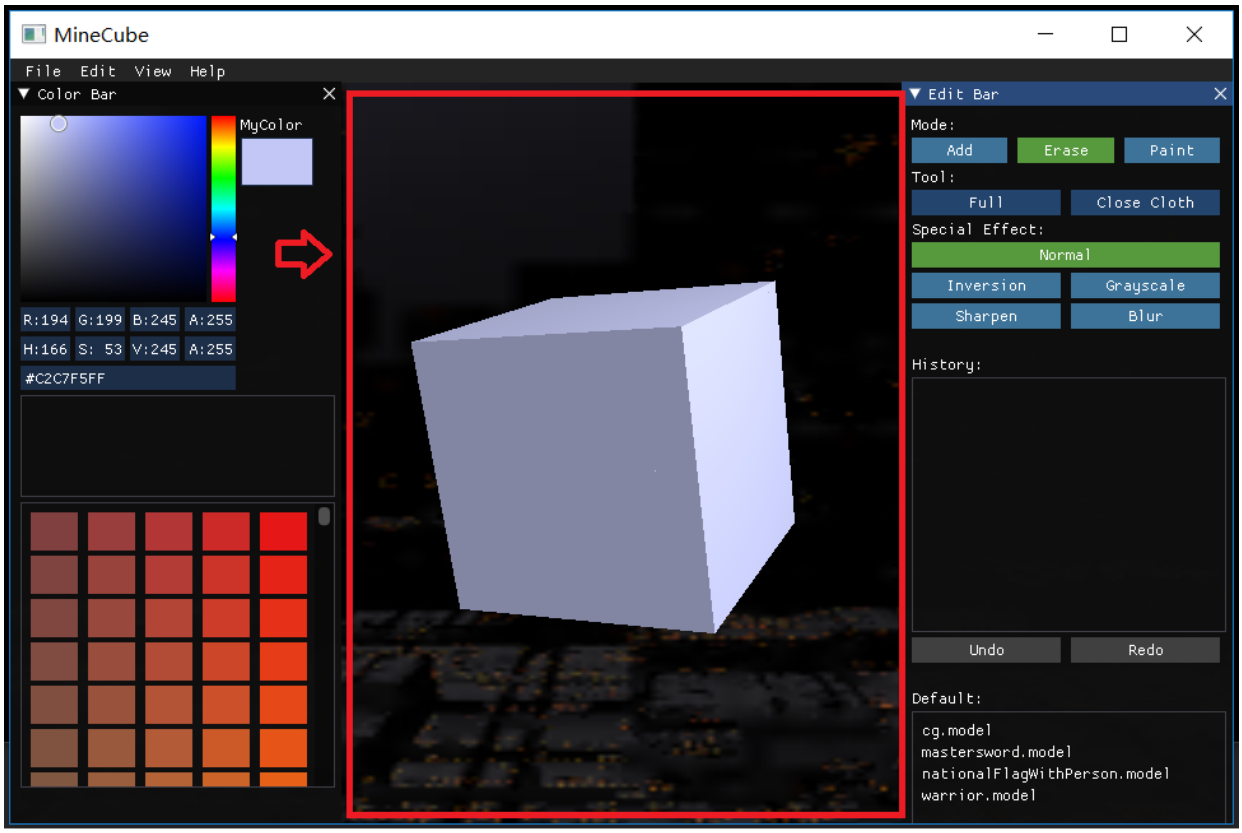


Sky Box

用相对简单的立方体贴图的方式实现。

立方体贴图就是一个包含了6个2D纹理的纹理，每个2D纹理都组成了立方体的一个面：一个有纹理的立方体 。

实验效果如图，在立方体的背景就是天空盒的实现效果：



（在摄像机漫游下可以转动摄像机进一步观察到天空盒的整体效果）

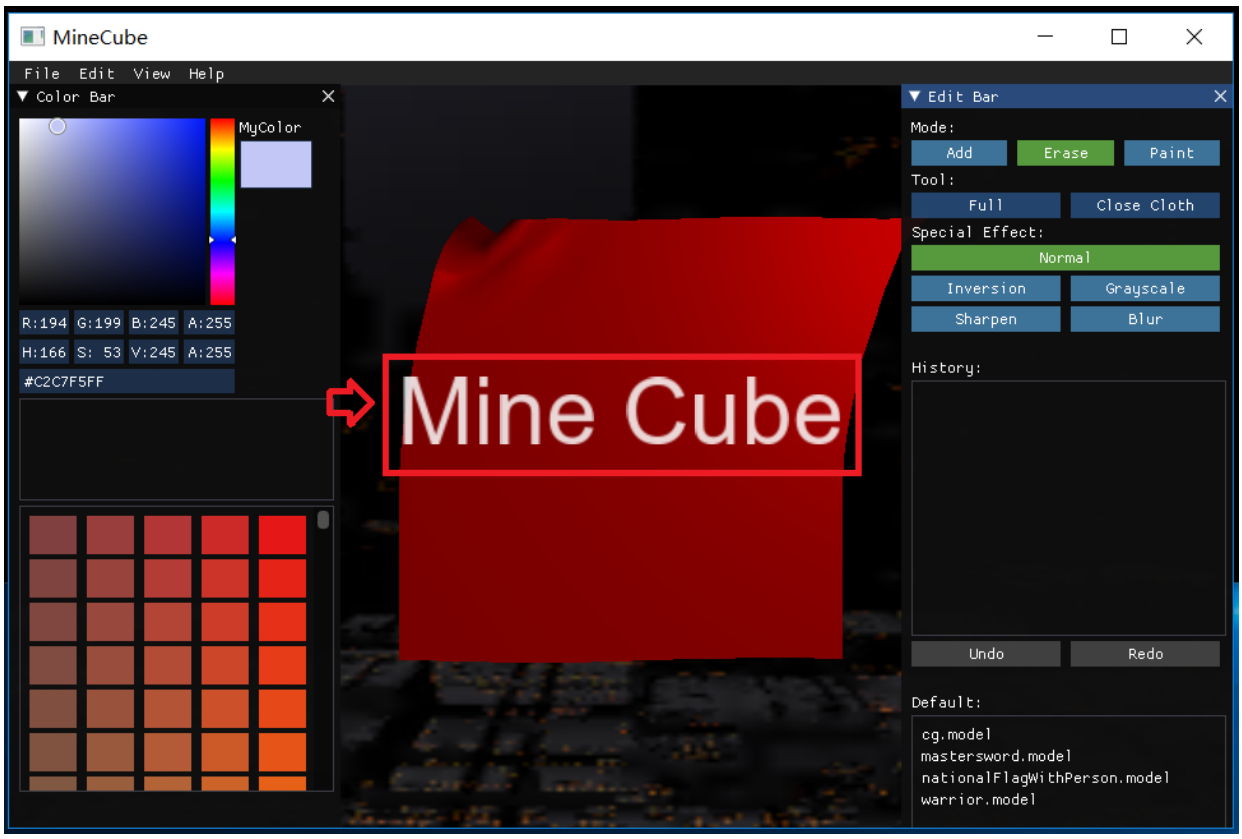
Display Text

使用现代对文本渲染方法，主要利用了 `freetype` 库（一个能够用于加载字体并将他们渲染到位图以及提供多种字体相关的操作的软件开发库）导入字体。

之后渲染时绑定纹理并进行了贴图混合：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

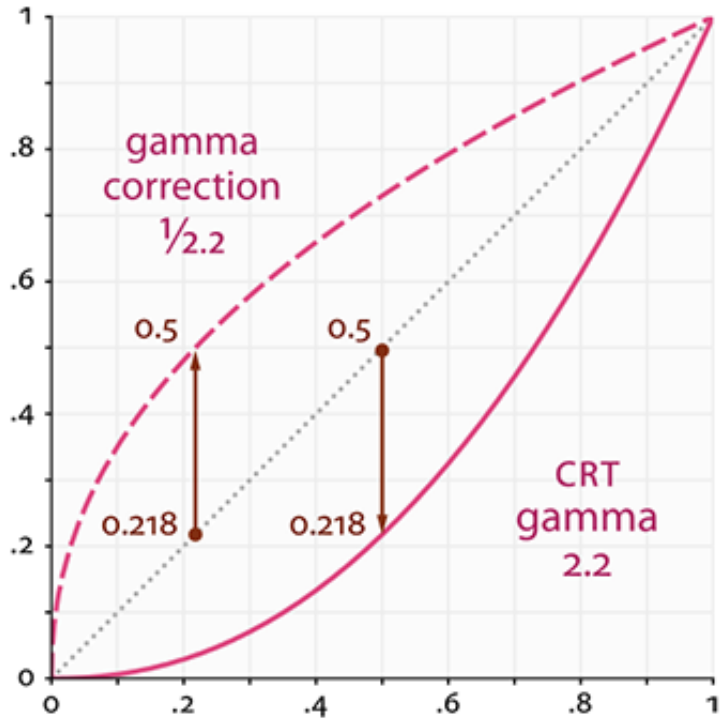
在开始时有显示软件名称并渐隐的效果，实验截图如下：



（在切换 Mode 时也会显示相应的名称并渐隐，效果类似，同上）

Complex Lighting （复杂光照: Gamma矫正）

人眼看到颜色的亮度更倾向于顶部的灰阶，监视器使用的也是一种指数关系（电压的2.2次幂），所以物理亮度通过监视器能够被映射到顶部的非线性亮度，这个非线性映射的确可以让亮度在我们眼中看起来更好，但当渲染图像时，会产生一个问题：我们在应用中配置的亮度和颜色是基于监视器所看到的，这样所有的配置实际上是非线性的亮度/颜色配置。

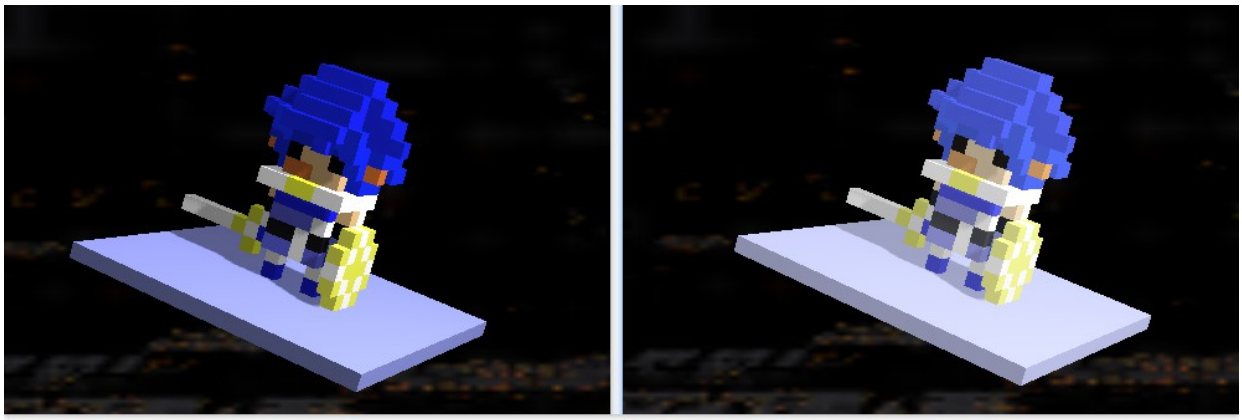


Gamma 校正(Gamma Correction)的思路是在最终的颜色输出上应用监视器Gamma的倒数。再看上面的Gamma 曲线图，你会有一个短划线，它是监视器Gamma曲线的翻转曲线。我们在颜色显示到监视器的时候把每个颜色输出都加上这个翻转的Gamma曲线，这样应用了监视器 Gamma 以后最终的颜色将会变为线性的。我们所得到的中间色调就会更亮，所以虽然监视器使它们变暗，但是我们又将其平衡回来了。

只需要在 Blinn-Phong shader 的片段着色器最后加上下面代码即可以实现 Gamma 校正。

```
void main()
{
    // do super fancy lighting
    [...]
    // apply gamma correction
    float gamma = 2.2;
    fragColor.rgb = pow(fragColor.rgb, vec3(1.0/gamma));
}
```

实验结果如下：



Cloth Simulation

采用弹簧质点模型，用**粒子系统**的方式对质点的速度、受力、位置进行模拟，其中受力也包含了**重力系统**。

弹簧质点模型是利用牛顿运动定律来模拟物体变形的方法，该模型是一个虚拟质点组成的网格，质点之间用无质量的、自然长度不为零的弹簧连接。其连接关系有以下三种：

1. 连接质点 `[i, j]` 与 `[i+1,j]`，`[i, j]` 与 `[i,j+1]` 的弹簧，称为结构弹簧
2. 连接质点 `[i, j]` 与 `[i+1,j+1]`，`[i+1,j]` 与 `[i,j+1]` 的弹簧，称为剪切弹簧
3. 连接质点 `[i, j]` 与 `[i+2,j]`，`[i, j]` 与 `[i, j+2]` 的弹簧，称为弯曲弹簧

这三种弹簧分别用于与结构力（拉力或压力）、剪力和弯矩相关的计算（弹簧的弹性力遵从Hooke定律）。

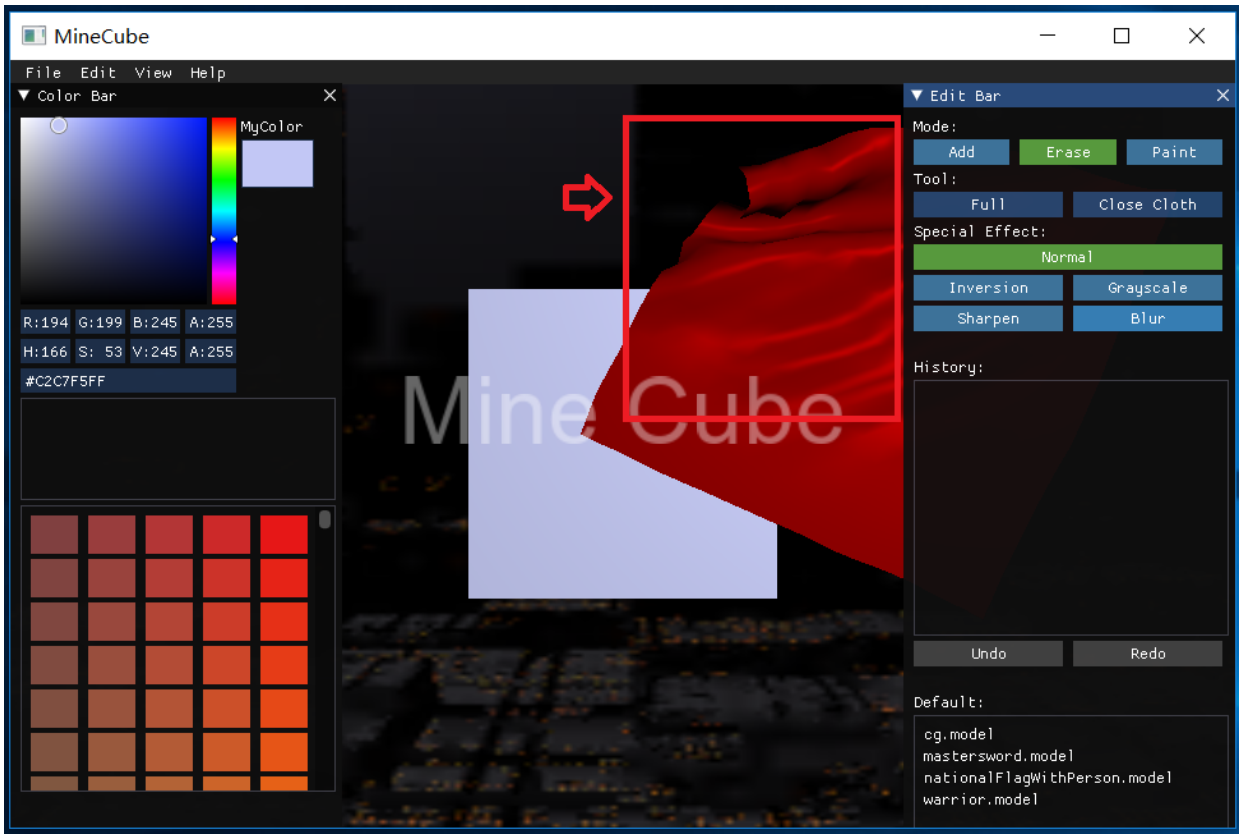
弹簧质点运动时受到内力和外力和影响，内力包括弹簧的弹性力和阻尼力，外力包括重力以及空气阻力等。

一个简要的核心代码如下：

```
void Cloth::simulate(float stepSize) {
    // CODE ...

    for (int i = 0; i < meshResolution; i++) {
        for (int j = 0; j < meshResolution; j++) {
            glm::vec3 newVelocity = getVelocity(i, j) + getForce(i, j) *
stepSize / mass;
            setVelocity(i, j, newVelocity);
        }
    }

    // Notice that the updated velocity above is used for better numerical
stability.
    for (int i = 0; i < meshResolution; i++) {
        for (int j = 0; j < meshResolution; j++) {
            glm::vec3 newPosition = getPosition(i, j) + getVelocity(i, j) *
stepSize;
```

```
                setPosition(i, j, newPosition);
            }
        }

        // CODE ...
    }

    glm::vec3 Cloth::getForce(int i, int j) {
        glm::vec3 F_spring = getAllSprings(i, j)
            + getGravityForce(i, j)
            + getDampingForce(i, j)
            + getViscousForce(i, j);
        return F_spring;
    }
```

在软件启动开始时模拟了幕布拉开的效果，实验截图如下：



参考文献：

- Baraff D, Witkin A. Large steps in cloth simulation[C]// Conference on Computer Graphics & Interactive Techniques. 1998:43-54.

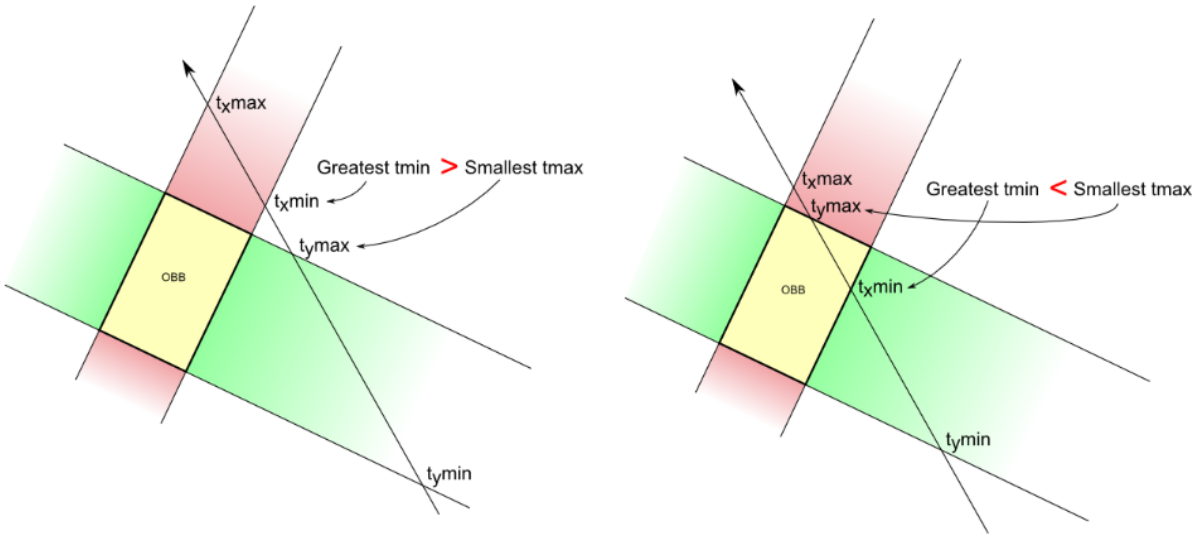
3D拾取

使用`Ray-OBB`的方法进行拾取。分为以下3步：

1. 将camera射线变换到世界坐标系。

2. 求射线与物体相交的面和距离。
3. 得到最短距离，产生这个距离的即是拾取到的方块。

射线与 `OBB` 的碰撞检测方法：根据射线进入和离开该物体的顺序来判断。如下，分别是射线**不穿过**和**穿过**的情形：

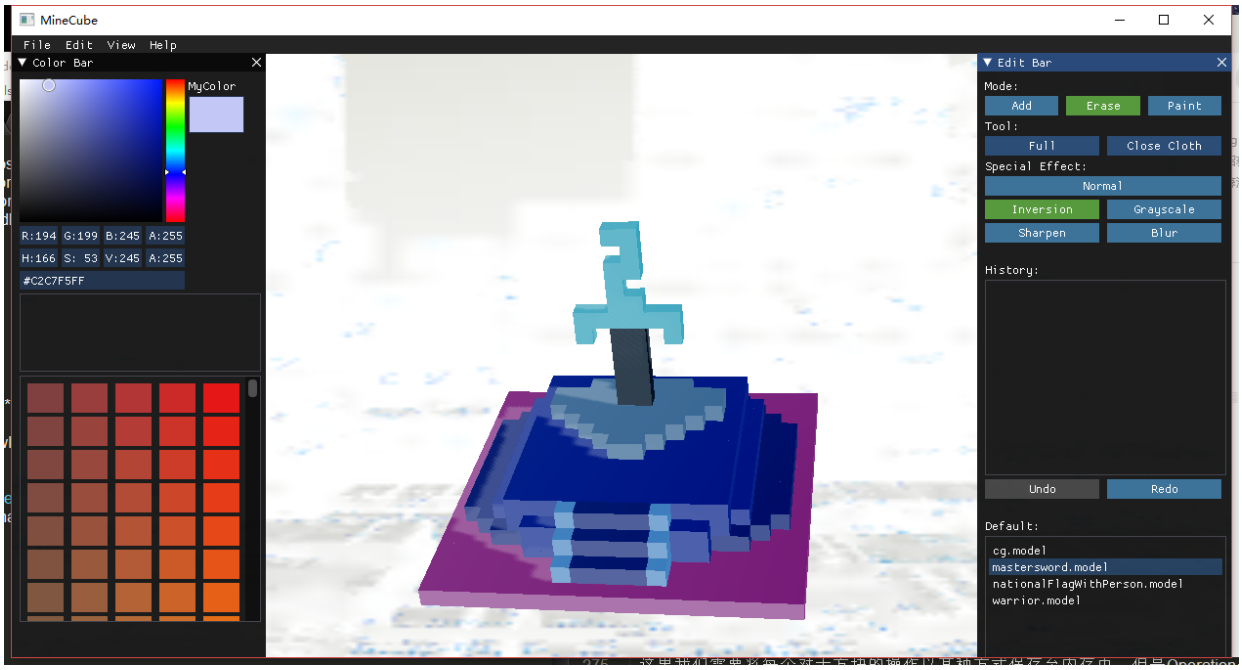


在所有被射线穿过的方块中，距离最近的（即最前面的）方块就是被拾取的方块。

注：如果首次运行时该功能出现 3D 拾取异常，原因是初始化的窗口的大小大于运行的机子屏幕的分辨率，随意改变一次窗口尺寸即可恢复正常。

帧缓存特效

将整个场景都渲染到了一个纹理上，我们可以简单地通过修改纹理数据创建出一些后期处理特效，例如反相：



遇到的问题和解决方案

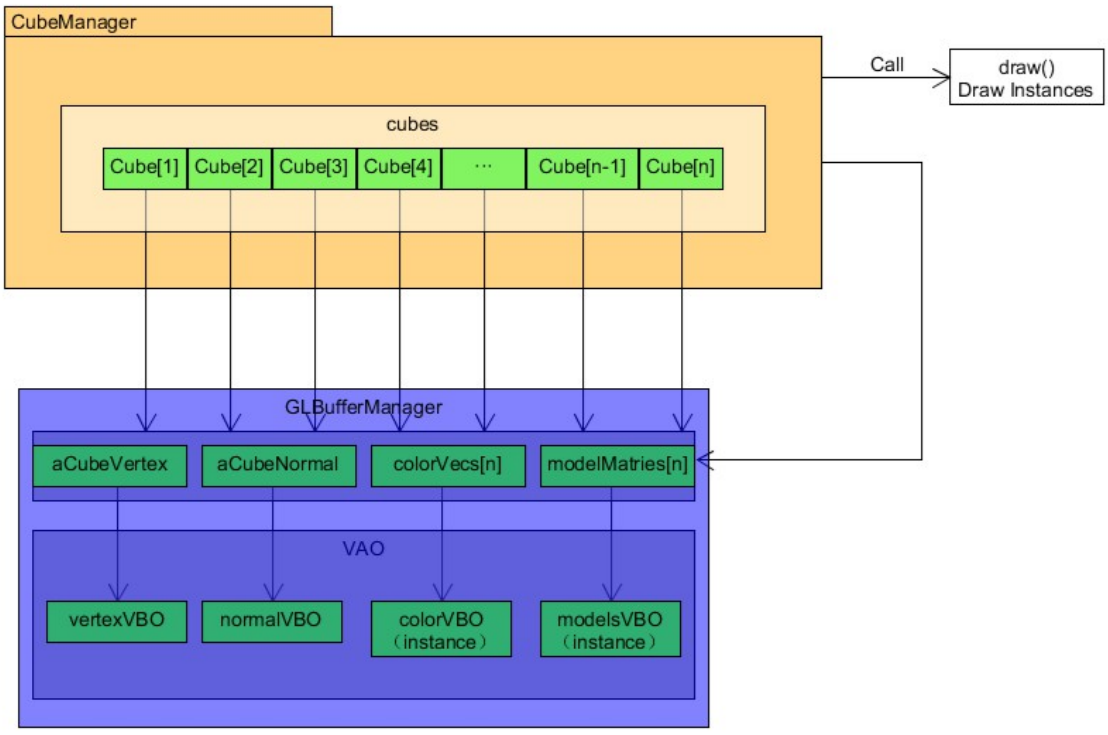
- 撤销操作的实现 这里我们需要将每个对于方块的操作以某种方式保存至内存中，但是Operation多种多样，很难定义一个普适的模型去保存Operation。我们这里利用了C++11中的`lambda`表达式去定义一个Operation，通过一个**execute**的`lambda`表达式和一个**undo**的`lambda`表达式完整定义一个操作，再将其保存至一个`stack`中，完成了撤销操作的需求。

  ```
  BasicOperation(const function<void()> & execute, const function<void()>
  & undo);
  ```

- 大量立方体渲染导致的渲染性能下降

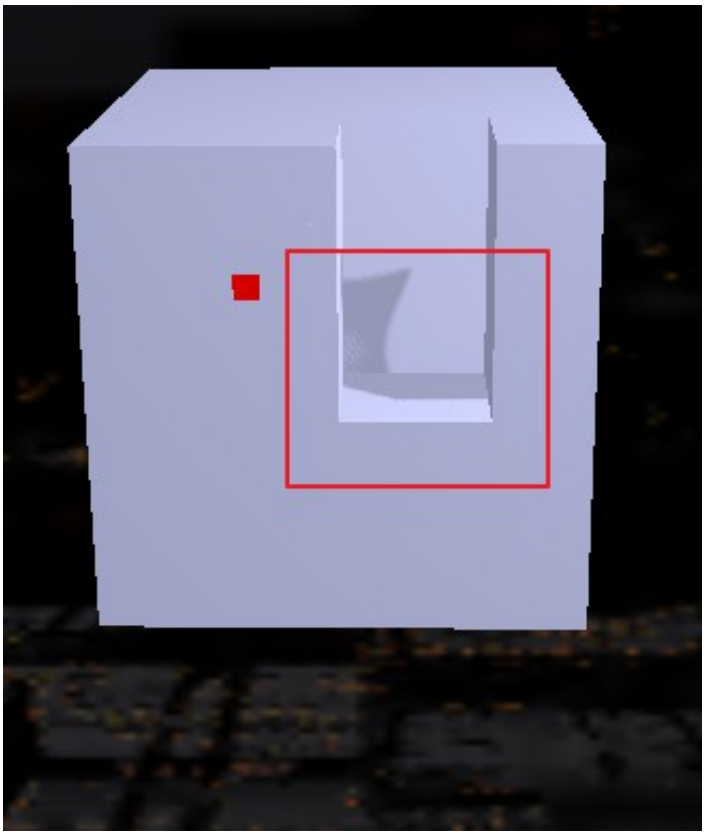
  我们的初始方案是让每一个小立方体独自渲染自己，我们很快就会因为绘制调用过多而达到性能瓶颈，最多只能渲染出一个 10x10x10 的大立方体，这样远远不能达到创作的要求。与绘制顶点本身相比，使用 glDrawArrays 或 glDrawElements 函数告诉 GPU 去绘制你的顶点数据会消耗更多的性能，因为 OpenGL 在绘制顶点数据之前需要做很多准备工作（比如告诉GPU该从哪个缓冲读取数据，从哪寻找顶点属性，而且这些都是在相对缓慢的 CPU 到 GPU 总线上进行的）。所以，即便渲染顶点非常快，命令GPU去渲染却未必。

  我们采用了是**实例化(Instancing)**的计算机图形学方法，将数据**一次性发送给GPU**，然后在顶层（CubeManager）使用一个绘制函数让 OpenGL 利用这些数据绘制多个物体，从而极大地提升了渲染的效率。目前项目可以流畅渲染 20x20x20 的立方体，当编译出来的是 release 版本的程序的时候， **30x30x30 的大立方体也可以流畅支持。**

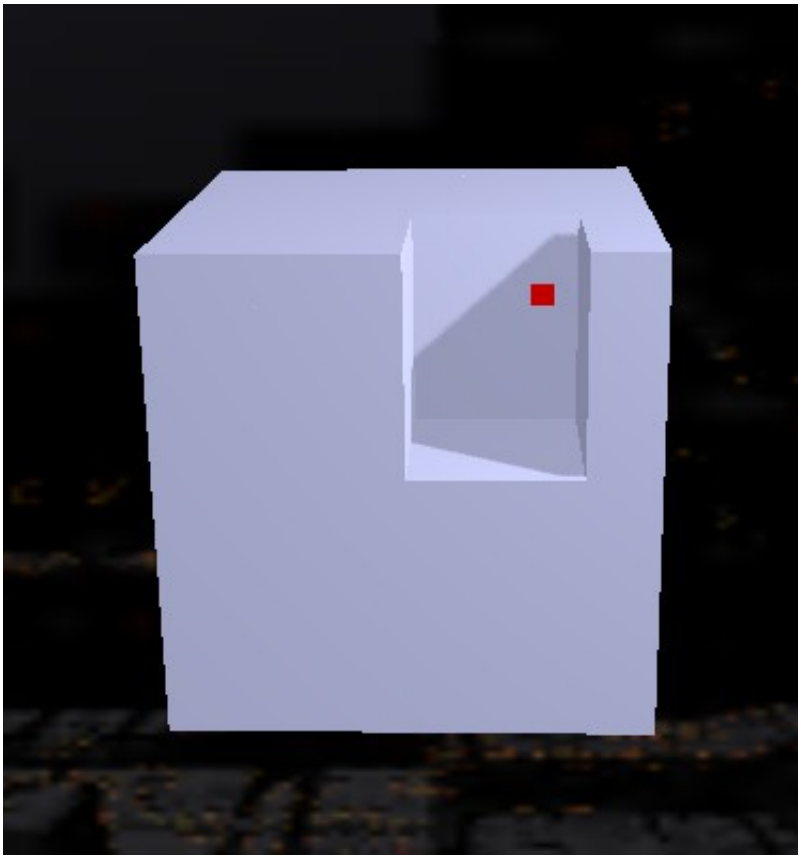


- 阴影支持 bias 过大

  使用作业的 Phong shader 的时候，因为小立方体之间的间隔较小，shader 中的 bias 设置得过大 (0.005) , 使得一些部分的渲染因为 bias 过大而认为没有处于阴影之中。



将 bias 适当调小后让阴影可以正常渲染。



- 着色器的调试
  在进行一些纹理贴图时容易出现渲染问题，在调试时难以追踪渲染管线中对数据变化。解决办法是使用 renderDoc 软件进行单帧捕获，从而调试渲染过程中的调用与数据问题。

小组成员分工

- 罗剑杰 @Johnny Law
  - Cmake 配置
  - Shadow Map 实现
  - blinn-phong 光照实现
  - Gamma 校正
  - 底层逻辑设计
  - 实例化渲染
  - 面剔除
- 吴博文 @Bob Wu
  - 初版底层模块构建
  - 方块增删改查
  - 模型导入导出
  - 撤销操作的实现
- 王治鋆 @Jarvis
  - Camera Roaming 实现
  - Shader 实现
  - 上层基本CRUD操作
  - 3D拾取
- 邱兆丰 @mgsweet
  - GUI实现
  - Undo和Redo上层实现
  - 上层批量CRUD优化
  - 帧缓存特效
  - 阴影效果修复
- 徐海洋 @Hiyoung.Tsui
  - 织物模拟（粒子系统）
  - 文本渲染
  - 天空盒（纹理映射）