编译实习期末报告

TDML-S-CSS

周子凯 1400012702　　钟原 1400012706

目录

1 选题描述

OpenGL 是一套跨语言、跨平台的 3D 图形编程接口。目前，OpenGL 是计算机 3D 图形编程的业界标准，其功能强大而完善，在各个相关领域被广泛应用。

但是，我们认为 OpenGL 有一个不足之处，在于它编写的程序冗长、细节繁杂、可读性差，而且学习门槛高，对初学者不友善。在 2D 图形绘制领域，虽然并未成为业界标准，但是 HTML+CSS+JavaScript 这一套组合在网页绘制上表现出了强大的易用性，其功能完善，扩展性强，程序更简洁易读，而且易于上手。这一套组合的应用已不局限于网页程序的开发，目前已出现了诸如 Electron 的桌面端 HTML+CSS+JavaScript 的应用开发平台，可见其受欢迎程度之高。

因此，我们希望在此大作业中进行一次尝试：仿照 HTML、CSS 和 JavaScript 的基本设计理念，设计一门能够描述 3D 场景的标记语言，以及与之配套的层级样式表和脚本语言，并开发将这三个语言组合成的程序编译到 C++语言的 OpenGL 工程，使之能够在不同平台上编译运行。我们将我们设计的标记语言取名为 TDML（Three-Dimensional Markup Language），层级样式表称为 mini CSS，脚本语言称为 mini JS。

2 设计思想

TDML、CSS、mini JS 三个语言共同组织成一个三维场景。其中，TDML 是程序的主体，是必须的，而 CSS 和 mini JS 是可选的。

仿照 HTML 的语法，设计出三维场景标签语言 TDML，用于描述三维场景中物体的布局以及属性。不同的标签表示了不同物体，而标签上的不同属性描述了物体的不同属性，如它的颜色、大小、位置以及其它更复杂的属性。就像 HTML 中的 DOM 树一样，标签之间的嵌套表述了物体之间逻辑上的从属关系，使得零散的物体间产生了“父元素”和“子元素”的关系，构成了树形结构。这便于用相对值来描述物体间的空间关系，如相对位移、相对大小、相对旋转等。将多个简单变换复合起来，更方便用标签语言描述三维场景。

沿用网页设计中 CSS 的语法，提取出一部分精髓语法设计出 mini CSS。TDML 中的标签属性可以写在标签上，也可以写在一个单独的 CSS 文件里。mini CSS 的存在可以使三维场景中物体的布局和属性可以分离开来，使代码更清晰易读。TDML 中可以指定每个物体属于的类别（class）或独有的 id，在 mini CSS 中可以统一地描述某一类物体共有的属性，例如为所有标签名为 x 的标签，或是所有 class 为 y 的标签，亦或是某个 id 为 z 的标签赋予某些属性。同时，mini CSS 也支持层级标签选择，如：为所有 class 为 x 的标签的子元素中标签名为 y 的标签统一赋予某些属性。

仿照 JavaScript 的语法，设计了一门新的类似 JavaScript 子集的脚本语言，称为 mini JS。mini JS 的存在是为了实现使用脚本语言动态地修改场景中的物体及其属性，最终实现动态三维场景编程。它实现了弱类型面向对象程序设计模式，支持最基本的运算、条件转移、循环、函数定义等功能，同时可以使用已实现好的内置函数完成一些复杂的功能，如在物体构成的树中寻找具有指定 id 的物体，或修改一个物体的某个属性等。

在上述三个语言都基本成型后，我们最终实现了将这三个语言的代码共同编译成 C++语言的 OpenGL 工程的编译器。该编译器的工作流程大致如下：

1.预配置好 OpenGL 工程，在工程中实现相关类和函数；
2.在 OpenGL 工程中准备好场景渲染用代码，用 lex+yacc 实现 TDML 和 mini CSS 的词法、语法分析，将 TDML 和 mini CSS 代码编译成兼容 OpenGL 工程中预留的渲染代码 API 的 C++代码；
3.在 OpenGL 工程中准备好实现了模拟弱类型面向对象机制的类，将 mini JS 代码翻译成使用该类实现的 C++代码；
4.将上述编译/翻译结果连同预备好的工程代码按特定的方式组织成一个完整的 OpenGL 工程，可直接在 Visual Studio 中编译运行。

3 TDML

3.1 描述

我们设计的 TDML 语言基本沿用了 HTML 的语法，不同的是我们为 TDML 赋予了重新设计的标签和属性。

由于 TDML 标签和属性种类繁多，对每个标签和每种属性的详细介绍参考《TDML 语言用户手册》。

3.1.1 标签

一个标签用于表示三维场景中的一个物体，或者用于控制代码的逻辑（如设置摄像机的初始位置）。

标签可分为两类，控制标签和图元标签。控制标签用于控制整个标记语言代码的逻辑，图元标签用于在场景中绘制图元。

控制类标签：tdml, head, body, camera

图元类标签：cuboid, sphere

标签的名称决定了该标签在文档中的基本功能。简要介绍每个标签的功能：

tdml：整个标签文档的根标签。除此之外无实际意义。

head：该标签的子标签用于控制代码逻辑（目前的实现中只有 camera 标签），标签本身不包含任何信息。

body：该标签的子标签用于描述场景内的物体，标签本身不包含任何信息。

camera：该标签的属性用于指定视角在三维空间中的初始位置和方向。

cuboid：该标签代表三维场景中的一个长方体。

sphere：该标签代表三维场景中的一个球体。

3.1.2 属性

标签可以带有属性。标签的属性用于进一步指示该标签的行为，或描述该标签的细节信息。标签属性描述的信息一部分是只与标签本身有关的“绝对信息”，如：cuboid 标签的 type 属性（样式）、color 属性（颜色）和 texture 属性（纹理）；另一部分属性描述的是相对于父

元素的“相对信息”，如 cuboid 标签的 x-rotation 属性（在父元素坐标系中自身坐标系的 x 轴的旋转角度）；有一些属性既可以以相对值的方式描述，也可以以绝对值的方式描述，如 cuboid 标签的 x-offset 属性（在父元素坐标系 x 轴方向上相对于父元素的位移），既可以指定为一个绝对长度，也可以指定为一个百分数，表示相对于父元素在 x 轴方向上的长度的百分之若干，还有 x-length 属性也是类似如此的一个属性。

图元的定位涉及到坐标系的概念。各个元素的基本坐标系是以物体中心为原点，平行于父元素坐标轴建立的正交坐标轴。如果指定了旋转相关的属性，则还需要将基本坐标系分别绕自身的 x、y、z 轴旋转相应的角度，得到元素的最终的坐标系。根元素的坐标系是相对于整个三维空间的坐标系建立的。

另外，标签有两个特殊的属性：tdmlid 和 tdmlclass，分别用于指定一个标签的 id 和所属的类（class）。这两个特殊属性用于配合 mini CSS 使用，将在 mini CSS 一节中详细介绍。

3.2 示例

一个基本 TDML 标签示例如下：

```
<cuboid x-offset="3" y-length="50%" color="rgb(1,0,0)" type="solid"></cuboid>
```

它表示一个长方体，在 x 轴方向上相对于父元素的位移是 3 个单位距离，y 方向上的长度是父元素 y 方向上的长度的 50%，颜色为红色，绘制方法为实心长方体。

如果一个标签不再有子标签，则可以使用不成对标签的写法。上例中的标签用不成对标签的写法写为：

```
<cuboid x-offset="3" y-length="50%" color="rgb(1,0,0)" type="solid" />
```

一组嵌套而成的标签形如以下代码：

```
<cuboid tdmlid=”father”>
    <sphere class=”son” />
    <cuboid class=”son” />
</cuboid>
```

它表示 id 为"father"的 cuboid 标签下有两个子标签 sphere 和 cuboid，它们都属于类"son"。

3.3 编译器实现

使用 Lex 和 Yacc 对 TDML 源代码进行词法分析和语法分析，语义分析部分交由 OpenGL 工程完成。

Lex 中，将标签名、属性名和属性值提取成 token，便于 yacc 程序书写。Lex 中的关键代码如下：

```
NAME -> [a-zA-Z_][-a-zA-Z0-9_]*
VALUE -> \"[^"]*\"
```

其中 NAME 表示标签名和属性名的 token，VALUE 表示属性值的 token。在 Lex 所有处理忽略所有空白字符，包括空格和\t、\n。

Yacc 中，识别出每个成对标签的开始标签、结束标签以及不成对标签，提取出其标签名、属性名-值对组成的链，并以上述信息为参数产生回调函数。Yacc 中的关键代码如下：

```
S -> S comlabel | comlabel
comlabel -> < label > | <label /> | </ NAME>
label -> label property | NAME
property -> NAME = VALUE
```

其中非终结符 comlabel 表示一个完整的开始标签、结束标签或不成对标签，非终结符 label 表示标签名+若干对属性名-值对，非终结符 property 表示一个属性名-值对。

Yacc 中完成了 TDML 标签组成的树结构的建立。维护一个保存从根标签到当前标签路径的标签栈。将不成对标签拆分成没有子元素的开始标签、结束标签，对每个开始、结束标签产生回调函数，每当遇到一个开始标签就向栈中压入一个标签，遇到一个结束标签就在弹栈的同时检查标签是否匹配。

在回调函数中，Yacc 生成一段 c++代码，输出到临时文件中。这段代码在树形结构中为当前标签新建节点、保存属性，将该节点加入到其父元素的子元素链中。该临时文件的内容最终被加入到 OpenGL 工程的指定位置中去。

3.4 细节和优化

3.4.1 如何存储属性

在存储属性时，由于各个属性的表示方法不同，占用内存大小不一，不能简单地用一个通用的结构体存储。并且，同一个属性可能有多种表示方法。例如，x-rotation 的属性值为一个实数，color 的属性值需要三个浮点数表示 r,g,b 三个通道上的值，而 texture 属性的值为一个长度不定的字符串；x-offset 属性可以是绝对值也可以是百分数，x-rotation 属性可以用角度制表示也可以用弧度制表示。这时候就需要设计一种高效、通用的方法存储各种各样的属性。我们设计的方法为：

为属性保存 3 个域，name、type 和 value。其中，name 表示属性名，type 表示这个属性使用了哪种表示方法，value 指向一块 void*类型的内存，用 type 指定的表示方式存储属性值。在建立新属性时，根据这个属性的空间需要动态分配内存，再将属性值存入这块内存区域。

读取属性时，首先读取 type 域的值，确定该属性是用哪种方式表示的，再用不同的方式解析 value 域的值。

这样做可以保证存储空间 0 浪费，同时也满足了能够存储不定长属性值、不同属性表示方法的需求。

3.4.2 对纹理属性的特殊处理

3.4.2.1 纹理优先级问题

cuboid 标签拥有纹理属性，用来指定长方体的 6 个面分别要显示什么样的图片。用户可以用 texture 属性统一指定 6 个面的公用纹理，也可以用 texture-x-positive 等 6 个属性分别指定长方体 6 个面各有的纹理。并且，为了方便用户使用，如果希望指定只有 x 轴正方向上的面的纹理与另 5 面不同，用户可以先指定一个 texture 属性，再单独指定 texture-x-positive 属性，而不需要把 6 面的纹理单独拆分开来指定。这里对两个属性的先后顺序并无要求。

为了实现这一点，程序中为纹理相关的属性特别设置了优先级，texture 属性的优先级比 texture-x-positive 等 6 个单面纹理属性的优先级要低。

3.4.2.2 如何方便地动态修改纹理

在 mini JS 一节中会提到，用户可以通过 mini JS 脚本代码动态地修改标签的属性，包括纹理。我们希望实现这一机制：虽然 texture 属性的优先级比 texture-x-positive 低，但是当用户通过脚本修改 texture 时，仍然能将整个长方体的六个面的纹理统一修改为指定的纹理，即便之前已经通过 texture-x-positive 之类的属性特别指定了某些面的纹理。

为了实现这一点，为纹理属性设计一套机制：在一个 cuboid 标签上，只存储了 6 个内部属性值，分别用于表示 6 个面上的纹理。但是，能够影响这 6 个内部属性值的有 7 个“外部属性”（即用户可以通过 TDML 属性、CSS 属性或脚本指定的属性及其值）：统一指定 6 个面纹理的 texture 属性，以及 texture-x-positive 等 6 个单独指定一个面的纹理的属性。为 6 个内部属性各自设置一个懒标记，当纹理外部属性被 mini JS 脚本更改时，设置这些懒标记为“被修改”：texture 属性将 6 个懒标记都做标记，而 texturte-x-positive 等 6 个单面纹理属性只修改其对应的 1 个懒标记。在渲染一个标签时，检查这 6 个懒标记，对于被标记为“被修改”的内部纹理属性，才修改它的值，并将懒标记清零。

4 mini CSS

4.1 描述

mini CSS 沿用了 CSS 的语法，用于将 TDML 中的标签和属性分离开来。mini CSS 存在的最大优点是，可以批量描述一类标签的共同属性。对于一个 CSS 规则，先用选择器指定该规则中的属性应该应用于哪些标签，再在规则体中书写属性本身。CSS 中的属性用法和 TDML 中的属性完全一样。

选择器用于选择 TDML 文档中的一类标签。选择器可以单独使用，也可以将多个选择器组合起来使用。

单一选择器：

共三种，分别为类选择器、id 选择器和标签选择器。

类选择器：.myclass，匹配属于 class 类的标签，即 tdmlclass 属性值为"myclass"的标签；

id 选择器：#myid，匹配 id 为 myid 的标签，即 tdmlid 属性值为"myid"的标签；

标签选择器：mytag，匹配标签名为 mytag 的标签。

组合选择器：

将多个单一选择器用空格连接，匹配的元素需满足：从树根到该元素的路径上依次出现了匹配每个单一选择器的元素，且匹配最后一个单一选择器的元素是其本身。

将多个组合选择器用逗号连接，可以实现多个组合选择器匹配同一条 CSS 规则。

4.2 示例

一个基本的 mini CSS 规则如下所示：

```
.apple
{
    color: "rgb(1,0,0)";
    radius: "1";
}
```

它使用了单一选择器.apple 选中了所有类别为 apple 的标签（即，属性 tdmlclass 的值为"apple"的标签），并统一为它们赋予属性 color="rgb(1,0,0)"和 radius="1"。

一个组合选择器的示例如下：

```
/*mini CSS 选择器*/
#two .big sphere, .small
{
    /*一些属性*/
}
```

应用到如下 TDML 文档片段中：

```
<cuboid class=”small” id=”one”>
    <sphere id=”two”>
        <cuboid class=”big” id=”three”>
            <sphere class=”big” id=”four” />
            <cuboid class=”small” id=”five” />
            <cuboid class=”big” id=”five” />
        </cuboid>
    </sphere>
</cuboid>
```

该选择器表示，选择所有 id 为 two 的元素下的属于 big 类的元素下的 sphere 标签，以及所有属于 small 类的标签。那么被该选择器选中的有 id 为 one，four 和 five 的三个标签。

4.3 编译器实现

使用 Lex 和 Yacc 对 mini CSS 源代码进行词法分析和语法分析，语义分析部分交由 OpenGL 工程完成。

Lex 中，将属性名、属性值和类选择器、id 选择器、标签选择器打包成 token，供 Yacc 使用。Lex 中的核心代码如下：

```
NAME -> [a-zA-Z_][-a-zA-Z0-9_]*
CLASS -> [.][a-zA-Z_][-a-zA-Z0-9_]*
ID -> [#][a-zA-Z_][-a-zA-Z0-9_]*
VALUE -> :[^;]+[;]
```

其中 NAME 表示标签选择器和属性名的 token，VALUE 表示属性值的 token，CLASS 表示类选择器的 token，ID 表示 id 选择器的 token。在 Lex 所有处理忽略所有空白字符，包括空格和\t、\n。

Yacc 中 ，识别出每个 CSS 规则的选择器以及它对应的属性名-值对组成的链。CSS 规则由若干个由","分隔的组合选择器构成，每个组合选择器又由一个或多个由" "分隔的单一选择器构成。Yacc 中的核心代码如下：

```
S -> S block | block
block -> nameset_ { propset }
nameset_ -> nameset_ , namelist_ | namelist_
namelist_ -> namelist_ threetype | threetype
threetype -> NAME | CLASS | ID
propset -> propset NAME VALUE | NAME VALUE
```

其中，非终结符 block 代表了一个 CSS 规则。每匹配到一个 block，就生成一段 C++代码，输出到临时文件中，用于在 OpenGL 中构建 CSS 列表。该临时文件的内容最终被加入到 OpenGL 工程的指定位置中去。

4.4 细节和优化

4.4.1 CSS 如何引入到 TDML 中

为每个 CSS 规则建立一个数据结构。为每个标签维护一条 CSS 规则链，依次链接它能匹配的每个 CSS 规则。在初始化 DOM 树或 DOM 树的结构发生变化时，遍历 DOM 树，更新每个标签的 CSS 规则链。

4.4.2 优先级问题

因为一个标签的属性既可以写在标签中，也可以写在与之关联的 CSS 规则中，因此就会引入当属性发生冲突时的优先级问题。沿用 HTML 和 CSS 的优先级规则，标签本身的属性优

先级高于 CSS 指定的属性。因此计算一个标签的最终属性时，时先应用 CSS 规则，再应用自身规则，这样可以使得后者的规则覆盖前者。

4.4.3 CSS 规则匹配效率问题

由于组合选择器机制的存在，一个 CSS 规则可能非常长。并且，标签的 id 和类是可变的，一个简单的改变可能引起整个标签树中每个标签的 CSS 规则匹配情况发生极大的变化。这引入了另一个问题：如何高效地判断一个 CSS 规则是否匹配一个标签？

可以采用如下贪心匹配算法：

DFS 整棵标签树的同时，为每个组合选择器 $Selector_i$ 维护一个各自匹配指针 p_i，表示从根到当前节点的路径上，该组合选择器最多能匹配前多少个单一选择器。在访问子节点时，根据每个指针 pi 指向的下一个单一选择器的信息以及该子节点的信息，可以确定该子节点是否能够匹配 pi 指向的下一个单一选择器，如果能，则移动指针到下一个单一选择器，并在此次递归前纪录这次移动，在回溯时还原。如果某个时刻，指针 p_i 已经移动到了组合选择器 $Selector_i$ 的末尾，且当前标签能匹配 $Selector_i$ 的最后一个单一选择器，则说明 $Selector_i$ 匹配了此标签。

从组合选择器功能设定出发，不难验证上述匹配算法能正确地找出所有匹配某个组合选择器的所有标签。并且，用上述算法确定每个标签是否需要匹配某个选择器时，不需要重新扫描一遍根到当前节点的路径，提升了匹配效率，尤其是在树高很大且组合选择器长度也很长的情况下。

5 mini JS

5.1 描述

仿照 JavaScript 的语法，取其语法的精华部分，设计出了 mini JS 脚本。mini JS 脚本的存在，使得用户具有动态修改 TDML 中每个标签的属性的能力，从而实现动态三维场景的绘制。

因为实现一个弱类型脚本语言到 C++的编译器需要大量的工作量，而实现 mini JS 编译器仅是本次大作业的一小部分，因此在语法和功能设计上，mini JS 秉持尽量精简的原则，通俗地说，刚好能满足需求就好。

5.1.1 变量

mini JS 是弱类型语言，用户通过"var"关键字申明变量，而无需指定变量的类型。而在编译器内部实现中，内部拥有的基础数据类型有三种：int, double, string。

申明变量时未赋值的变量默认值为(int)0。

变量可以通过被赋值来改变其内部存储的值的类型。

为了避免未申明变量造成的麻烦，要求所有全局变量需要在程序的开头申明，所有函数内的局部变量必须在函数体的开头申明。

5.1.2 对象与数组

JavaScript 中一大特色是其对象的设计。在 mini JS 中同样实现了对象。用户可以通过 var a=new Object();来申明一个名为 a 的对象，然后给对象 a 添加属性。属性可以动态添加，添加属性之前不需要事先申明属性。并且，对象的属性可以是另一个对象。如，在上述申明了对象 a 的代码后，可以用代码 a.x=1;为对象 a 赋予属性 x，其值为 1。具体使用方法参考“示例”一节。

数组在编程语言中是必不可少的一个功能。mini JS 中，把数组看成特殊的对象，将对象和数组统一实现为了对象，认为数组元素只不过是一个对象属性名为整数的属性。用"[]"访问属性名是整数的属性（即当作数组用时的元素访问），用"."访问属性名是字符串的属性。

5.1.3 函数

用户可以用如下语句申明一个函数：

```
function myfunc(var1, var2, var3)
{
    /*函数体*/
}
```

myfunc 是函数的函数名，var1~var3 是传入的函数参数。因为 mini JS 是弱类型语言，所以不需要指定参数和返回值的类型。mini JS 函数使用语句 return x 返回一个值。mini JS 的函数必须有返回值，如果用户不需要一个函数的返回值，可以简单地返回 0 或者其它任何值。

JavaScript 中，支持在函数中申明新的函数，同时将函数作为特殊的对象看待，可以参与赋值、参数传递、返回值等，实现各种复杂的功能。mini JS 不支持在函数体中申明函数，也不支持将函数作为一个对象参与赋值、参数传递、返回值等。

5.1.4 运算符

mini JS 支持 JavaScript 语言的绝大部分运算符，同时由于 JavaScript 的运算符大多与 C++相同，mini JS 的运算符的用法与 C++基本没有区别，下面便不再赘述各个运算符的细节，只列出 mini JS 支持的所有运算符。

1. 数值运算符（int, double）
   +，-，*，/（浮点数除法），&，|，^，>，<，>=，<=，==，++， --， =（赋值）
2. 字符串运算符（string）
   +，=（赋值）
3. 布尔运算符
   &&，||

运算符的优先级与 C++相同。可以用圆括号改变运算的优先级。

5.1.5 基本语法

除了上述变量、对象、函数申明以及变量运算，mini JS 支持基本的控制流语法，包括：

- if 语句
- for 语句
- while 语句

以上三种语句语法与 C++对应的语句语法完全一样，便不再累述。

因为 mini JS 没有 bool 类型，因此在涉及条件判断的地方，将空字符串和数字 0 视为 false，其余视为 true。

5.1.6 API

mini JS 存在的最重要目的是，与 TDML 文档中的标签交互，从而动态地修改文档内容。为了实现这一点，需要语言内置函数的支持。同时，为了能让最终 3D 程序的使用者和 3D 场景进行交互，需要开放能获取当前鼠标、键盘状态的内置函数给用户。

在 OpenGL 工程中实现了若干用于与 TDML 文档和输入设备交互的函数，并为用户开放以下 API：

`function getElementById(myid)`

返回标签树中 id 为 myid 的标签对象。该对象可以作为 modify 函数的 element 参数被传入。

`function isKeyboardDown(key)`

返回当前时刻键盘上的按键 key 是否为按下状态。key 为一个 0~255 的整数，表示要查询的键的 ASC II 码值。

`function isMouseDown(key)`

返回当前时刻鼠标上的按键 key 是否为按下状态。key 为一个 0~2 的整数，0 表示左键，1 表示中键，2 表示右键。

`function Cos(x)`

返回 cos(x)的值。在实现视角旋转时很有用。

`function Sin(x)`

返回 sin(x)的值。在实现视角旋转时很有用。

`function modify(element, propertyName, propertyValue)`

将标签树中的 element 元素的名为 propertyName 的属性的值设为 propertyValue。如果该属性不存在，则会创建该属性。其中，element 参数必须是一个由 getElementById 函数返回的对象。

动态场景的渲染是逐帧进行的。为了实现随着时间的推移，不断地执行代码查询和修改 TDML 文档中的信息，用户可能需要在每帧渲染开始之前执行一些代码。同时，用户也可能需要在渲染开始的第一帧之前执行一些代码做准备工作。为了实现以上两个机制，mini JS 保留了两个函数，需要用户来实现它们：

`function onRedisplay()`

此为一个留给用户实现的函数。此函数的函数体为在每帧渲染开始之前都会执行一次的代码。此函数在每帧渲染开始前自动被系统调用。

`function mainScript()`

此为一个留给用户实现的函数。此函数的函数体为渲染开始的第一帧之前会执行一次的代码。此函数只会被在程序运行的开始被调用一次。

从以上 API 的描述中不难得出每个 API 的每个参数的值类型。由于 mini JS 遵循尽量简单的设计原则，向 API 函数传入类型不正确的参数均为未定义行为。

5.2 示例

可以用如下语句初始化三个变量 a,b,c：

```
var a, b=1, c="xxx";
```

执行此语句后，由于 a 没有指定初始值，因此 a 的值为默认值 0；b 的值为数字 1；c 的值为字符串"xxx"。

可以用如下语句初始化一个对象：

```
var a=new Object();
a.b =1.0;
a[1]=new Object();
```

执行上述代码后，对象 a 拥有两个属性"b"和"1"。a.b 的值为浮点数 1.0；a[1]的值为一个空对象。

假设 TDML 文档中有如下定义的 camera 标签：

```
<camera tdmlid="camera" y-offset="-20" />
```

以下代码实现了用键盘按键'w'控制视角前进的功能：

```
var cam = new Object();
var camDOM;
var velo = 0.1;

function onRedisplay()
{
    if (isKeyboardDown('w'))
    {
        cam.pos.y = cam.pos.y+velo;
    }
    modify(camDOM, "y-offset", cam.pos.y);
    return 0;
}

function mainScript()
{
    cam.pos = new Object();
    cam.pos.x = 0;
    cam.pos.y = -20;
    cam.pos.z = 0;
    camDOM = getElementById("camera");
    return 0;
}
```

其中，代码开头申明了全局对象 cam 和 camDOM，cam 用来保存当前视角在三维空间

中的位置和指向的方向，在 mainScript 中被初始化；camDOM 在 mainScript 中被初始化成指向 TDML 文档中的 camera 标签的对象；velo 表示移动速度。在程序开始运行时，函数 mainScript 被调用一次，初始化 cam 对象和 camDOM 对象。在每帧渲染之前，onRedisplay 函数被调用，函数中首先检查了键盘上的'w'键是否被按下，如果是，则将本地维护的 cam 对象的 y 坐标加上 velo；之后，通过 modify 函数更新 TDML 文档中 camera 标签的 y-offset 属性值为本地维护的 y 坐标值。视角在位置移动的过程中，方向始终朝向原点(0,0,0)。

5.3 编译器实现

mini JS 的语法与 C++相似性很高，不同之处在于申明变量、对象和函数所用的关键字。可以用静态翻译的方法将 mini JS 代码“改写”为 C++代码，在源代码中发现特定模式的时候（如一个函数申明），就将该模式改写成 mini JS 语法的写法。

但需要注意的是，mini JS 代码中用到的字符串常量和注释中可能出现正好匹配了 mini JS 特定模式的子串。因此，在扫描源代码时需要一边扫描一边维护当前代码处于字符串常量或注释之内还是之外。

将 mini JS 编译成 C++的一大问题是实现 mini JS 中的对象机制。mini JS 中的对象机制完全不同于 C++，使用对象前不需要提前申明该对象所属的类，对象还能够动态地新增属性。可以用 C++ STL 中的 map 来模拟 mini JS 的对象，因为 map 原生支持动态增加元素，且可以用字符串作为元素的键值。在内部实现中，将一个 map 包装成一个 Object 类，便于后续实现别的功能。

另一个问题是如何实现弱类型变量。为了实现弱类型机制，在 OpenGL 工程中设计了一个类__Type，它包含四个类型分别为 int, double, string, Object 的属性，分别用来保存一个 mini JS 变量保存的值的内部属性分别是整数、浮点数、字符串和对象时的值。一个 Object 对象被实现为一个 map<string, __Type>。在翻译 mini JS 代码时，一个由 var 关键字申明的变量翻译到 C++代码中就变成了一个__Type 类型的变量。

第三个问题是如何实现弱类型变量之间的运算。同一个运算表达式可能因为参与运算的弱类型变量的内部类型不同而得到完全不同的结果。解决方案是，在 OpenGL 工程中编写__Type 类型与 int、double、string 类型之间的强制类型转换，以及为各个运算的不同参数版本编写重载函数。这样就可以在保持源代码基本不变的情况下，通过重载函数内部判断参与运算的两个变量的当前内部属性来执行不同的运算。

5.4 细节和优化

5.4.1 垃圾回收

在 mini JS 中，对象采用引用传值的方法进行传递。这导致了一个问题：当一个函数的返回值是一个对象时，这个对象可能是该函数中的一个局部变量，但在返回值被传递给调用者后，该对象不应该被析构。因为调用者得到的返回值仅仅是一个对象的引用，它的属性值仍然存在于被调用者的局部环境中。

为了解决这一点，在 mini JS 的翻译结果中需要将用到的对象用 C++的 new 关键字创建。但这引入了另一个问题：每帧被刷新之前都可能有脚本被执行，如果不及时析构脚本创建出

的对象，将会使得内存占用量激增直至程序崩溃。

然而 C++本身不存在垃圾回收机制，需要用户在合适的位置手动析构全局对象。在一个以方便使用为目标的面向对象脚本语言中，要求用户手动析构对象是不现实的。为了解决内存占用量的问题，我们在 C++工程中实现的 Object 对象中实现了引用计数机制。当一个局部 Object 对象被 C++析构时，它可以在析构函数中判断其指向的各个属性的计数值是否已归零，如果是则可以将这个对象从对空间中抹去。

虽然引用计数不能完全解决内存泄露问题，但是对于以尽量简单为设计原则的 mini JS 来说，这点已经足够了。

6 测试集

见 sample 文件夹

css+tdml

物体属性同时受自身属性和 css 规则影响。用于测试 css 的有效性，以及规则的优先级关系

script

单独测试 mini js 脚本。用于测试弱类型机制、隐式类型转换、基本运算结果的正确性

css+tdml+script

将三者结合，综合测试，用于检测 script 动态修改元素属性的能力。

7 代码部分介绍

new 文件夹

解析 tdml 的 lex 和 yacc 程序，以及处理开始、结束标签回调和标签属性数据转移的 labelHandler.c 程序。

css 文件夹

解析 css 的 lex 和 yacc 程序，以及处理 css 规则数据转移的 cssHandler.c 程序。

script 文件夹

翻译 mini js 和转移翻译后代码的程序。

homework 1 文件夹

OpenGL 工程所在文件夹，以及最终结果工程的存放位置。工程中预先实现了各提供给 mini js 的类和函数。

compile.bat

用于编译整个工程的脚本。

8 完成情况

任务	完成情况
设计语言语法细节	完成
标记语言词法、语法分析（使用现有库）	完成
样式表词法、语法分析	完成
标记语言+样式表语义分析结果编译到 OpenGL	完成
脚本词法、语法分析	完成
结合脚本语义分析结果编译到 OpenGL	完成
移植 OpenGL 工程，测试、优化	优化未完成

9 分工

周子凯

语言（词法、语法、机制）设计，API 函数实现，三维场景绘制代码编写，文档编写

钟原

Lex、Yacc 代码编写，实现 mini JS 翻译器，文档编写

10 未完成的事项

- 图元可变基准点（实际应用价值不大）
- 动态加入 DOM 元素
- 载入模型
- 丰富 mini JS 的 API 函数库（目前需要靠用户手写 mini JS 脚本来实现一些更复杂的功能）