**建模软件开发**

**建模软件架构与开发方案选型**

建模软件预期要实现2个功能，模型参数的计算和模型的渲染预览。

模型参数的计算可以使用任何开发语言实现，即模型参数的计算功能对建模软件的开发方案没有直接影响。为了实现参数计算与效果渲染的互操作，这里优先进行三维渲染的技术选型，反过来再决定整个软件的技术选型。

首先排除使用OpenGL或DirectX等底层图形接口进行渲染功能的开发，因为使用底层图形接口需要考虑很多与硬件设备有关的技术细节，会拖累软件开发周期。因此这里选择使用成熟的三维渲染引擎。目前业界通常采用Unity3D或Unreal引擎进行三维场景的设计与开发，由于Unreal引擎更适用于游戏场景开发，与之对应的Unity3D引擎更为通用。本文所讨论的建模软件选择Unity3D引擎开发三维渲染。

完成三维渲染的技术选型后，考虑软件的具体实施方案。这里有两个方面需要单独说明。

一方面，对于预期的软件，模型的参数计算不在Unity3D里开发，而是抽取隔离到一个桌面软件中。这与一般的Unity3D开发方式不同。通常 Unity3D开发会通过Unity3D实现软件的所有功能。

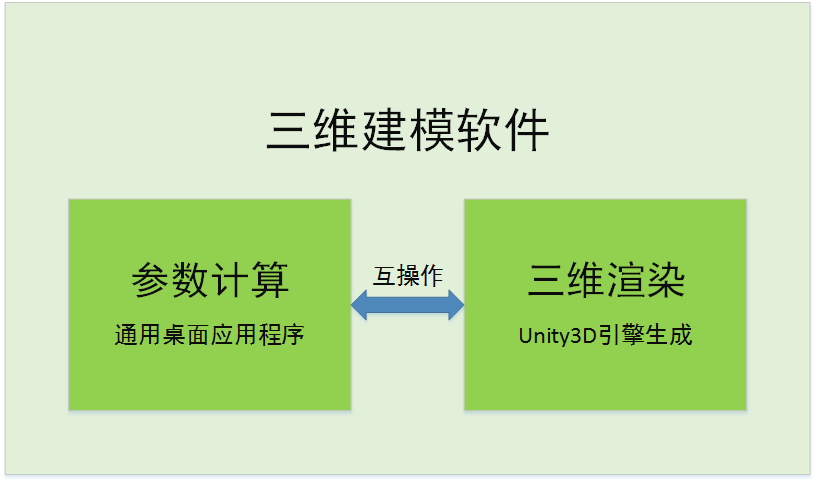
这样选择的原因是， Unity3D的架构约束了整个软件的架构，降低了整体软件的架构的自由度，而且Unity3D的业务逻辑通过脚本语言执行方式执行，性能较差。

另一方面，本文所述建模软件重新设计了三维的交互操作，将其替代为基于通用图形用户界面交互，同时实现三维人工刷新预览。这样做一定程度上降低了用户体验，但是这项弱化带来了软件开发的极大便利。

本文的建模软件是为了验证一种建模思想，并不是开发一款工业级三维建模软件。在这两个方面的约束下设计开发的建模软件，可以保证在较短的开发周期实现较多的功能，而且有效提升了模型参数的计算性能，提高了建模过程的自由度。

Unity3D引擎可以使用C#、javascript或Booscript作为脚本语言，桌面软件可以选择C/C++、C#等。这里选择使用C#作为建模软件两部分的开发语言，Unity3D引擎的C#使用Mono开源运行时，而Windows环境下桌面软件开发使用的C#语言以.NET Framework作为运行时。

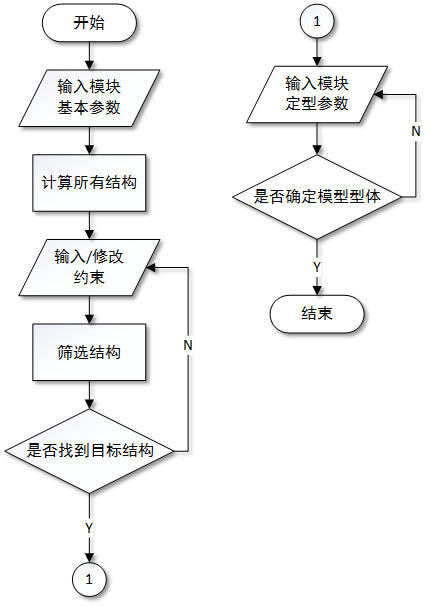
由上文讨论，软件的主体架构随着开发技术选型而确定。如图：



图题：三维建模软件基本架构

**三维模型的建模过程的软件流程**

如图是建模流程图。



图题：建模流程

首先，用户通过图形界面创建需要的模块。

然后，用户使用参数计算部分的结构计算功能，计算所有可能的模型结构。

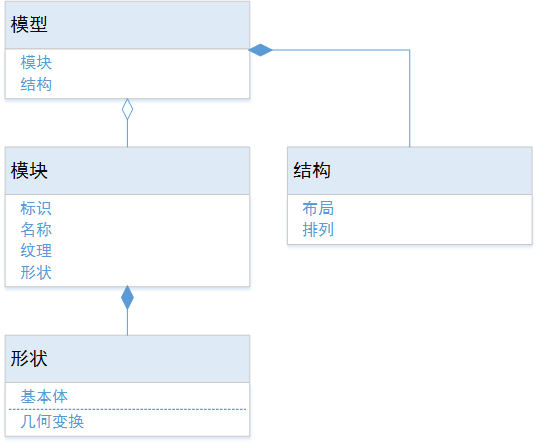
根据用户的先验知识，用户创建结构约束，通过结构过滤器的计算，筛选出所需的模型结构。模型结构可以通过三维渲染功能预览。如果筛选后结构数量仍然较多或预览后没有需要的结构，用户可以修改或增减结构约束，然后重新过滤筛选。如此迭代，直到根据预览的情况，用户确定需要的结构。

对于确定的结构，用户可以在模型定型图形界面进行模型定型配置。模型定型时用户可以为每一个模块指定基本体和三维变换参数。

保存定型的模型并通过三维渲染功能预览，如果需要微调参数可以在模型定型图形界面修改参数，保存并重新预览。如此迭代直到获得理想的参数，保存渲染图完成一次三维模型建模。

**数据结构设计**

如图是使用面向对象方法分析得出的模型类图。



模型由模块构成，描述一个完整模型，需要其构成模块、空间结构两部分信息。

模块具有标识、名称、纹理、形状等属性。模块形状属性包括一个模块的基本体、模块基于关键位点的几何变换两部分信息，其中几何变换包括相对关键位点的平移、相对原始尺寸的缩放和以坐标轴为转轴的旋转，这里使用笛卡尔右手直角坐标系作为几何变换的参考系。模块的纹理使用位图即可。

空间结构根据空间结构的生成算法，可以分为两步。第一步是确定结构的关键位点坐标，关键位点是安放模块的基础点位。第二步是确定模块与关键位点的对应关系。所以空间结构信息由一个三维坐标数组和一个坐标关于模块的字典构成，进一步简化可以使用一个三维坐标数组和一个模块索引数组描述空间结构。

**算法说明**

一个模型由N个模块组成。N个模块可以构成很多空间结构（具体数量见公式）。

首先穷举所有关键位点构成的布局，每一个布局由一组关键位点描述。

然后穷举N个模块放置在N个关键点位的全排列。

一个布局中关键位点之间存在一定的递推关系，可以用状态机描述。同时递推关系可以看做关键位点之间存在因果关系，这就使得布局的形成过程可以用树的数据结构表示。寻找所有布局是对树的末端进行一次完全遍历。这可以通过DFS（Deep First Search）深度优先搜索算法实现。

同样，排列的形成过程也可以用树结构表示，寻找全排列的过程就是排列树的末端的完全遍历。也可以通过深度优先搜索实现。

DFS具有多种实现，主要分为递归方法和非递归方法。使用递归方法，当N的数值较大时，递归深度较深，会急速耗尽进程的栈内存，进而导致进程崩溃。所以这里使用非递归DFS。非递归DFS通过使用额外的内存记录搜索状态，由于将递归过程转化为普通循环，只需要消耗堆内存即可，所以相对递归方法更稳定。

以下是两个关键搜索算法的伪代码。

**布局遍历算法**

*layouts* = new **layout** **list**

*N* = count of *modules*

*stack* = new **iterator** **stack**

*points* = new **iterator** **list**

*point* = new **iterator** as **{x:0, y:0, z:0}**

push *point* into *stack*

add *point* into *points*

while true {

if *points*.*Count* == N then {

*layout* = convert from *points*

add *layout* into *layouts*

remove *point* from *points*

pop *stack*

if *stack*.*Count* == 0 then break

*point* = peek from *stack*

*point* move next

} else {

If *point*.*NextPoint* == null then {

remove *point* from *points*

pop *stack*

if *stack*.*Count* == 0 then break

*point* = peek from *stack*

*point* move next

} else {

*point* = *point*.*NextPoint*

push *point* into *stack*

add *point* into *points*

}

}

}

**排列遍历算法**

关键部分：根据当前排列计算下一排列

[input] *arrange* is **int** **list**

*pos1* = 0, *pos2* = 0, *find* = 0

*end* = *arrange*.*Count* - 1

*pos1* = *end*

while *pos1* != 0 {

*pos2* = *pos1*

*pos1*--

if *arrange*[*pos1*] < *arrange*[*pos2*] then {

*find* = *end*

while *arrange*[*find*] <= *arrange*[*pos1*] {

*find*—

swap *arrange*[*find*] and *arrange*[*pos1*]

reverse *arrange* from *pos2* to end

return Success

}

}

reverse *arrange* from pos1 to end

return Fail

[output] *arrange*