# UE4.26运行时的Mesh生成与编辑

# 背景

- 本文作者Ryan Shmidt是著名的C#开源库Geometry3Sharp[1]的作者，随着G3Sharp库的不断迭代改进，其功能变得愈发强大，然而C#的性能问题以及C#生态中缺乏稀疏线性解算器等关键数学库等因素成了其继续发展的瓶颈。于是，作者在2018年12月决定加入Epic Games，成为Epic Games的首席几何学家（Principal Geometry Scientist），并着手将G3Sharp移植到C++。伴随着作者和虚幻引擎几何开发组的同仁们的辛劳付出，如今虚幻引擎C++版本的几何模型处理插件（Geometry Processing Plugin）的功能已经远远超过了C#版本的G3Sharp库。作者认为时机成熟，于是写下了一系列的技术博客来介绍如何在虚幻引擎中使用这个强大的几何模型处理库。

- 上周我们发布了系列教程中的第一篇《在虚幻引擎4.26命令行程序中处理网格模型》（点击蓝字查看）。下文便是系列教程中的第二篇，本篇翻译对其略作精简。

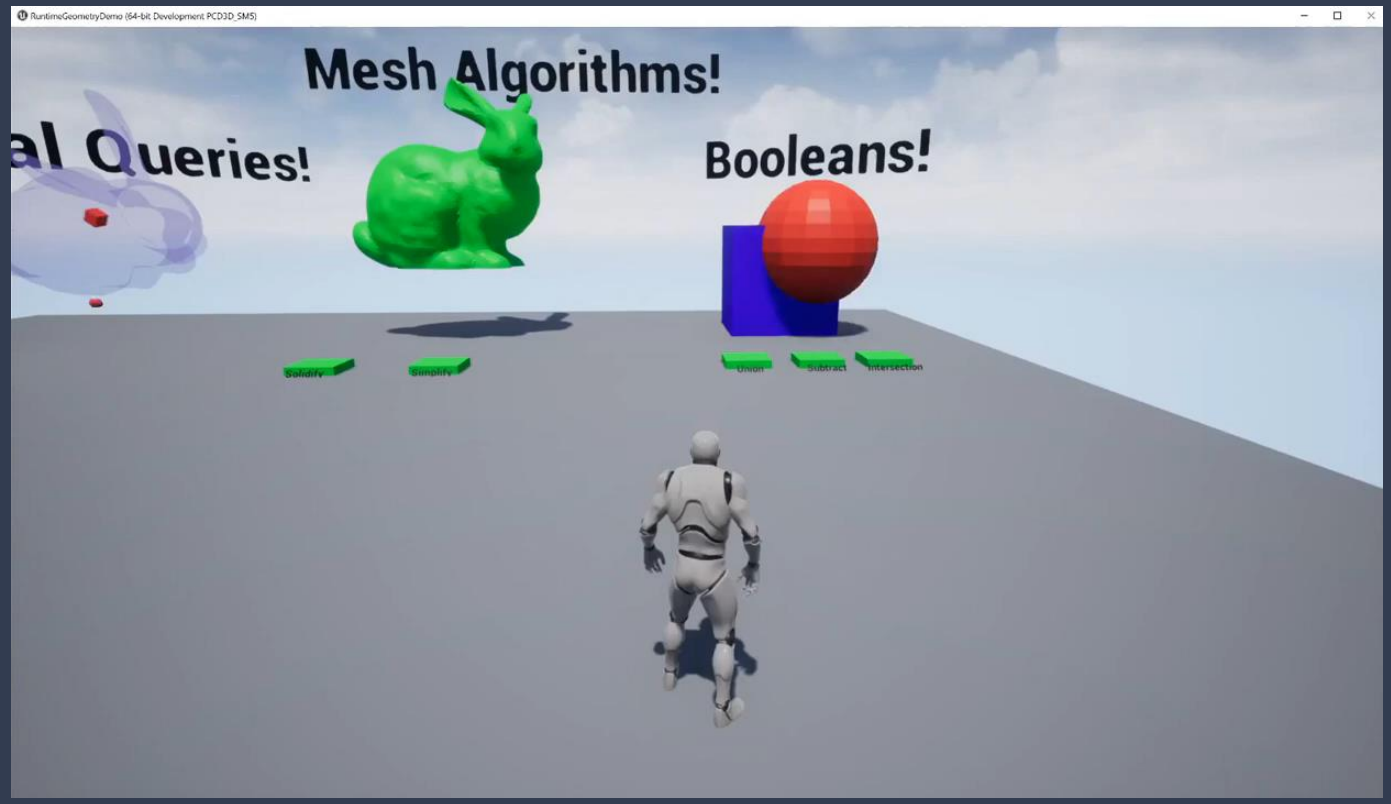
- 原文链接：http://www.gradientspace.com/tutorials/2020/10/23/runtime-mesh-generation-in-ue426

## 正文

- 在我的上一个教程中，我展示了如何使用UE4.26中新的实验性GeometryProcessing插件执行各类实用的网格模型操作，如网格生成、重新网格重建，简化和网格布尔运算（zomg！）当人们第一次了解到我们在虚幻中添加了这些功能时，他们眼前的问题通常是"我可以在游戏（运行时）中使用吗？答案就一个字“是”。然而，这个插件并没有暴露蓝图API，所以为了做到运行时使用，还有些小坑要趟掉。

- 一个经常出现的问题是如何在UE4游戏或应用程序中实现运行时网格创建。在过去，大家主要使用UProceduralMesh组件（API文档链接[2]）来实现。但自打UE4.25起，UE已经可以在运行时“生成”和更新UStaticMesh，然后这些UStaticMesh可以在UStaticMeshComponent/Actor中使用。所以现在的问题变成了你应该用哪一个？此外，还有第三方解决方案（如RuntimeMeshComponent[3]），它们提供的功能比UProceduralMesh组件更多，在某些情况下可能是更好的选择。（在本教程的其余部分，UProceduralMesh组件缩写为PMC，UStaticMesh组件缩写为SMC）。

- 不幸的是，没有”最佳”选项——这取决于你的需求。并且，如何将其和我们的GeometryProcessing插件结合使用的方法并不那么显而易见，GeometryProcessing插件使用FDynamicMesh3来表示网格，而和Actor以及Component没有任何关系。因此，我将在本教程中展示实现运行时生成和编辑Mesh的一种方法。



## 正文

- 上方的视频展示了一个简单的“运行时几何编辑”演示，这个演示使用了我在这个教程中实现的Actors和工具。如你所见，有布尔运算、网格操作和一些空间查询的功能演示。这些只是我通过蓝图暴露的一部分功能，在本教程结束时，你会发现通过我下面将介绍的ADynamicMeshBaseActor类来暴露其他基于几何处理插件的网格编辑代码非常简单。

## 获取和运行示例项目

- 在我们开始之前，本教程是UE4.26的，当前处于预览版（撰写本文时处于4.26 Preview4）。你可以通过Epic Games Launcher安装4.26的最新版本。

- 本教程的项目位于GitHub上的UnrealMeshProcessingTutorials代码仓库（MIT许可证[4]） 中的UE4.26/RuntimeGeometryDemo子文件夹中。一个坏消息是此项目目前只能在Windows上工作。但好消息是通过一些选择性的删减，它应该能跑在OSX或者Linux上的。我将在文章末尾介绍怎么做。如果你不想使用git获取示例项目，你可以通过以下链接直接下载项目的zip包。

- https://github.com/gradientspace/UnrealMeshProcessingTools/archive/master.zip

# 获取和运行示例项目

- 进入RuntimeGeometryDemo最外层文件夹后，右键单击Windows资源管理器中的RuntimeGeometryDemo.uproject，然后从上下文菜单中选择“Generate Visual Studio project files”项目文件。这将生成RuntimeGeometryDemo.sln。你也可以直接在编辑器中打开.uproject（它会要求编译），但我想大概率你是需要参考下本教程的C++代码的。

- 生成解决方案并运行（按F5），UE编辑器应该会打开演示关卡。你可以使用主工具栏中的大大的Play按钮在PIE模式中测试项目，或者单击“Launch”按钮生成可执行文件。这会需要几分钟，之后生成的游戏将在单独的窗口中运行。在演示关卡里随便跑跑，向墙体射击！因为没有菜单/UI，你需要按Alt+F4来退出。

## 代码

- 本教程主要分两大部分。首先，我将描述一个用C++实现的在运行时动态/编辑网格Actors的架构，然后介绍如何在项目中使用这些Actors暴露的蓝图来执行有趣的操作。除了核心功能，没有任何“游戏逻辑”是用C++实现的。我已经把相关功能封装在了名为RuntimeGeometryUtils的插件里了，你可以很容易地复制到自己的项目。顺便提一下，在RuntimeGeometryUtils中，我也包括了我之前教程中DynamicMeshOBJReader/Writer的更新版本，但我已经将相关API修改为静态函数。

# PMC还是SMC?

- 如果要在游戏运行时构建动态几何体，第一个问题是到底用UProceduralMesh组件（PMC）还是UstaticMesh 组件（SMC）？两者之间有各种区别，但最主要的区别在于性能。若要在运行时更新网格，需要先“构建”网格。我的意思是需要创建或更新网格用于渲染的表现形式。虚幻不会直接用PMC中的Section数据来渲染，也不直接用SMC中的FMeshDescription进行渲染。对于任何UMeshComponent（PMC和SMC都是），需要创建一个FPrimitiveSceneProxy子类，它是组件用于渲染的表现形式。该代理将从组件数据创建一个或多个FMeshBatch。

- 这在 PMC 中相对直白些，我建议去看下ProceduralMeshComponent.cpp中实现它的那部分代码，FProceduralMedeSceneProxy类在文件头部。你会看到在其构造函数中FProceduralMeshSceneProxy在外部创建的FProcMeshSection做了转换并且初始化了FStaticMeshVertexBuffers和FLocalVertexFactory。这些结构保存了GPU渲染所需要的数据，如顶点位置、三角形索引缓冲区、法线和切线、UV、顶点色等。对于GPU来说，这些数据在PMC或SMC之间没有区别，唯一的区别是数据如何被添加到这些结构中。

# PMC还是SMC?

- SMC要复杂得多。我在这里用的术语不太严谨，因为UStaticMeshComponent不存储网格本身——它引用一个UStaticMesh，是UStaticMesh负责存储网格数据。在UE4.25之前，无法在运行时更新UStaticMesh。这是因为传统上，你的“源网格(SourceMesh)”在编辑器中以FMeshDescription的形式储存在UStaticMesh，它是通过“烘焙”生成的一个预处理，优化过的“渲染网格（rendering mesh）”,用它来初始化FStaticMeshSceneProxy（在StaticMeshRender.cpp中完成）。FMeshDescription在烘焙后不再被使用，因此在构建的游戏中会将它剥离。而这个烘焙过程，由UStaticMesh::Build()启动，依赖于各种仅能在编辑器模式下运行的函数和数据，这就是为什么你不能在运行时更新UStaticMesh几何体的原因。

- 但是，在4.25中添加了一个新函数——UstaticMesh::BuildFromMeshDescriptions()。此函数接受一组FMeshDescription作为输入并用其初始化渲染用的网格数据，这就允许我们在运行时生成UStaticMesh。运行时与编辑器模式下的生成路径不同——它跳过了在运行时执行速度过慢（例如生成距离场照明数据）或没有用处的各种复杂步骤（例如生成光照贴图UV，运行时无法烘焙光照贴图，所以这步在运行时毫无意义）。

# PMC还是SMC?

- 调用BuildFromMeshDescriptions()比在ProceduralMeshComponent更新Sections更昂贵。权衡一下，你可以在多个StaticMeshComponent中复用生成的UStaticMesh（获得Instanced Rendering带来的好处），而如果你想要具有多个"相同"的PMC，则需要将Mesh复制到每个PMC中。此外，你还可以使用 SMC 获得额外的渲染功能。例如，PMC中使用的FProcMeshVertex仅支持4个UV通道，但UStaticMesh最多支持7个UV通道。UStaticMesh还支持LOD和Sockets，通常SMC在整个引擎中受到更好的支持。

- 另一个根本的区别在于两者使用渲染器的方式。PMC使用所谓的“动态绘制（Dynamic Draw）”绘制，这意味着每帧它要重新生成并提交FMeshBatches。这基本上等同于告诉渲染器“我的顶点/索引缓冲区在任何帧上可能会更改，所以不必费心缓存任何东西”，这导致一个性能成本。SMC使用“静态绘制（Static Draw）”，该路径告诉渲染器渲染缓冲区不会改变，因此它可以进行更激进的缓存和优化。如果你对此感兴趣，你就千万别错过Marcus Wassmer在GDC2019上就UE4当前的渲染管线做的技术分享 https://www.bilibili.com/video/BV1Tb411L7tt?from=search&seid=11120763181825751244。

# PMC还是SMC?

- 这些并不是仅有的区别，但是就运行时生成的网格而言，这两点是需要权衡的核心区别。SMC将为你提供更好的渲染性能，但具有更高的前置成本。因此，如果你只是加载生成的网格，或者正在“完成”动态编辑或更改网格，则使用SMC构建网格将更有优势。但是，如果你需要不断（经常）修改网格，那么如果你使用SMC将不得不每帧做更昂贵的更新操作，并因此性能受到影响。作一个简单的测试，我使用下面描述的基础结构在“tessellation level 32”重新生成每帧的球体，该级别生成2046个三角形和1025个顶点。使用PMC，这在 PIE中以90-100fps的速度运行。使用SMC时，FPS下降到30。如果我将细分调高到128（32k三角形），它仍可以在使用PMC时达到约15fps，而使用SMC时，帧率是不可接受的3fps。

## 第三个选项：USimpleDynamicMeshComponent

- 不得不在两个组件中纠结已经让事情复杂化，让我再火上浇把油！PMC有个基础的限制，它无法分离顶点上的法线，UV或者其他顶点属性。所以如果你有一个拓扑上由连接的面构成的立方体，意味着每条边被面共享，每个角上的顶点有3个法线，UV，每个面有自己的UV Chart/Island，这种情况下你必须在初始化PMC之前将立方体分成6个独立的长方形，这么做的原因是PMC实现时希望在初始化RenderProxy时尽可能地高效，而GPU不支持任何数据拆分。顶点拆分和重写三角形序号很繁琐，这会让PMC的API变得十分复杂。所以你必须自己来拆分。

- 这也就意味着PMC不适合直接用于任何类型的网格编辑。例如，你想用一个平面去切割一个立方体并填充切割面。用PMC也就意味着你要直接切割PMC的Sections，你手里的数据实际上不是一个立方体而是6个不相连的三角形补丁。因此在切割后你得到的不是一个对于洞填充操作很理想的封闭边界环，而是得到一些需要连接起来才能定义这个洞的3D线段。这对编辑的可靠性非常不利。类似地，如果你“拉”这个立方体的某个角，你只会创建一个裂缝。我还可以列举更多……相信我，这会是场噩梦。

## 第三个选项：USimpleDynamicMeshComponent

- 因此，为了在UE4.25中实现中的网格建模编辑器模式，我们引入了另一种网格组件，USimpleDynamicMeshComponent。这个组件类似于PMC，因为它使用动态绘制（Dynamic Draw）路径，并且设计上就考虑了需要经常被更新的情况。然而，与PMC不同的是，它存储了一个更复杂的网格表示，支持在顶点处的属性拆分，并在内部处理了将网格数据重写为适合于GPU工作的格式。这个更复杂的网格表示是什么呢？自然是一个FDynamicMesh3。

- 为什么命名为“简单（Simple）”？因为还有另一种变体——UOctreeDynamicMeshComponent——它被设计用来高效地更新大网格的子区域。这超出了本教程的范围。

## 第三个选项：USimpleDynamicMeshComponent

- USimpleDynamicMeshComponent（简写为SDMC）有许多PMC所不具备的特性，这些特性非常适用于交互式网格编辑。例如，它支持对模型三角面的子集进行材质覆盖，或者隐藏它们，而不需要重新构造网格。它也有各种“快速更新”不同渲染缓冲区的功能。例如你只改变顶点的位置或者颜色或者法线。它还支持将网格“分块”到多个渲染缓冲区以快速更新，并且可以在网格更新时自动计算切线。但我并不打算在本教程中展示如何使用这些功能。

- 在取舍方面，SDMC将比PMC生成更大的渲染缓冲区。因此，如果GPU内存在你的动态几何编辑“游戏”是瓶颈，它可能不是最好的选择。不过，在交互网格编辑的情况下，这种内存使用与你将需要的许多网格副本和辅助数据结构相比，通常是微不足道的。SDMC目前也不支持任何物理解算。最后，它不能被序列化——你应该只在生成的网格会以其他形式保存的情况下使用SDMC。

# 运行时几何架构和ADynamicMeshBaseActor

- 好的，现在我们有3个选项——SMC、PMC和SDMC。它们都以不同的方式存储网格（FMeshDescription、FProcMeshSection和FDynamicMesh3）。我们应该选用哪一个？小孩才做选择题，我都要！

- 架构的核心问题是，生成的网格来自何方?如果你是完全程序化地生成网格，或者是从文件中加载，那么你可以很容易地使用任何一个选项，唯一的问题是，它在生成之后是静态的，还是需要经常更新，或者你想“偷懒”不去自己构建Sections。

- 如果在网格生成之后需要更改它们，那么我强烈建议你不要将这些组件中的网格表示看作你应用程序数据模型的规范表示。对于初学者，它们都不会序列化你的网格数据。除了网格顶点和三角形之外，你可能还需要跟踪其他元数据（即使你现在不需要，将来你很可能会需要）。所以，我认为你应该仅仅将不同的组件看作是渲染网格数据的不同方式。

## 运行时几何架构和ADynamicMeshBaseActor

- 在本教程中，“my mesh data”将存储为FDynamicMesh3。在我看来，如果你没有自己的网格，这是一个很好的选择。但是要知道FDynamicMesh3目前还没有本机序列化，你需要自己实现它。下一个问题是把这些数据放在哪里。我将在放在一个C++ Actor类中，ADynamicMeshBaseActor。如果我正在构建一个真正的应用程序，比如说一个网格雕刻工具，我可能会把网格放在其他地方，然后在它被修改时将它传递给ADynamicMeshBaseActor。但现在，我将直接在Actor里操作：

```
UCLASS(Abstract)
class RUNTIMEGEOMETRYUTILS_API ADynamicMeshBaseActor : public AActor
{
protected:
    /** The SourceMesh used to initialize the mesh Components in the various subclasses */
    FDynamicMesh3 SourceMesh;
};
```

# 运行时几何架构和ADynamicMeshBaseActor

◆ 这个Actor并没有显示网格的能力，它需要一个组件。我将创建三个Actor子类分别对应三种组件：

```
UCLASS()
class RUNTIMEGEOMETRYUTILS_API ADynamicSMCActor : public ADynamicMeshBaseActor

{

    UPROPERTY(VisibleAnywhere)

    UStaticMeshComponent* MeshComponent = nullptr;

};


UCLASS()
class RUNTIMEGEOMETRYUTILS_API ADynamicPMCActor : public ADynamicMeshBaseActor

{

    UPROPERTY(VisibleAnywhere)

    UProceduralMeshComponent* MeshComponent = nullptr;

};
```

# 运行时几何架构和ADynamicMeshBaseActor

```
UCLASS()
class RUNTIMEGEOMETRYUTILS_API ADynamicSDMCActor : public ADynamicMeshBaseActor

{

    UPROPERTY(VisibleAnywhere)

    USimpleDynamicMeshComponent* MeshComponent = nullptr;

};
```

◆ 现在，在ADynamicMeshBaseActor上，我们将加入以下函数，该函数没有实现（但不能声明为纯虚，即=0，因为UE在UObjects里不支持这么做）：

```
protected:
    /**

    *Called when the SourceMesh has been modified. Subclasses override this function to

    * update their respective Component with the new SourceMesh.

    */

virtual void OnMeshEditedInternal();
```

- 最后，我们在每个子类中实现这个虚函数。从本质上讲，每个函数必须做的就是从FDynamicMesh3生成和更新其组件的网格数据。在文件MeshComponentRuntimeUtils.h中，我添加了转换器函数来做到这一点，UpdateStaticMeshFromDynamicMesh()和UpdatePMCFromDynamicMesh_SplitTriangles()。前者使用FDynamicMeshToMeshDescription类（几何处理模块的一部分）执行转换。对于PMC，函数当前只是拆分每个三角形。从GPU内存的角度来考量，这不是最高效的，但在CPU端是最快的（在交互式网格编辑中，我们倾向于关注CPU端，因为这通常是瓶颈）。

- 对于SDMC，它只需要一个直接拷贝操作。请注意，SDMC支持直接编辑其内部的FDynamicMesh3。我本可以允许基类访问SDMC的内部网格，这可以在某些时候避免复制的开销。但是，我不会使用组件的成员数据作为我的规范源网格，我上面提过这是一个坏主意。在某些性能敏感的情况下，它可能有意义，但FDynamicMesh3的复制是非常快的，所以为了保持代码干净，我没有在这里这样做。最后，我们在ADynamicMeshBaseActor上有一个顶层API函数来实际修改原始网格：

# 运行时几何架构和ADynamicMeshBaseActor

```
/**
 * Call EditMesh() to safely modify the SourceMesh owned by this Actor.
 * Your EditFunc will be called with the Current SourceMesh as argument,
 * and you are expected to pass back the new/modified version.
 * (If you are generating an entirely new mesh, MoveTemp can be used to do this without a copy)
 */
virtual void EditMesh(TFunctionRef<void(FDynamicMesh3&)> EditFunc);
```

- 基本上，你调用此函数并传入一个C++ Lambda表达式来执行你实际的网格编辑操作。这种模式使我们能够更好地控制对原始网格的访问，因此理论上使得我们可以从其他地方“借用”它。另一个函数::GetMeshCopy()允许你从Actor中取得原始网格的副本，例如，在要组合两个网格的情况下需要该函数。

◆ 基本上就是这样。如果我们有上述3个ADynamicMeshBaseActor子类中的任何一个实例，我们可以在C++中更新它，执行以下操作如下：

```
SomeMeshActor->EditMesh([&](FDynamicMesh3& MeshOut)
{
    FDynamicMesh3 NewMesh = (...);
    MeshOut = MoveTemp(NewMesh);
});
```

◆ 底层的PMC、SMC或SDMC将自动更新。

- 这里你可能有一个合理的对设计的质疑是，为什么这一切都在Actor上完成，而不是在组件上完成？当然，它也可以在组件上完成。这其实取决于你打算怎么处理网格。UE的一个复杂情况是，一个Actor可以有许多组件，然后如果你想要执行诸如"组合网格"等操作，你必须决定如何处理多个（可能有层级关系）的组件，其中一些组件可能不是可编辑的网格。如果你的主要处理逻辑放在组件，Actor会怎么样？（这些都是我们在设计模型编辑器模块中仍在不断挣扎的概念问题！！）对于本教程，我想将每个网格视为一个“对象”，因此通过Actors进行组织是有意义的。此外，Actor有蓝图，这将更容易做下面这些有趣的事情。

# 网格生成

- 不幸的是除非我们在C++里初始化它，目前的ADynamicmeshBaseActor在编辑器下不会做任何事情。因此，我添加了一些基本的网格生成功能。顶层UProperty SourceType确定网格是使用生成的基元（框或球体）还是使用OBJ格式导入的网格来进行初始化。“执行组件”属性的相关部分如下所示（单击以放大）。我还添加了控制如何生成法线、为基础组件分配哪些材质以及生成和导入的其他选项的功能。此外，还有选项来自动构建AABBTree和FastWindingTree，我将在下面讨论。

# 网格生成

**Mesh Options**

| | |
|---|---|
| Source Type | Primitive |
| Normals Mode | Face Normals |

**Material Options**

| | |
|---|---|
| Material |  M_RedMaterial |

**Primitive Options**

| | |
|---|---|
| Regenerate on Tick | ☑ |
| Primitive Type | Sphere |
| Tessellation Level | 16 |
| Minimum Radius | 100.0 |
| Variable Radius | 50.0 |
| Pulse Speed | 3.0 |

**Spatial Query Options**

| | |
|---|---|
| Enable Spatial Queries | ☐ |
| Enable Inside Queries | ☐ |

**Mesh Options**

| | |
|---|---|
| Source Type | Imported Mesh |
| Normals Mode | Split Normals |

**Material Options**

| | |
|---|---|
| Material |  M_Translucent |

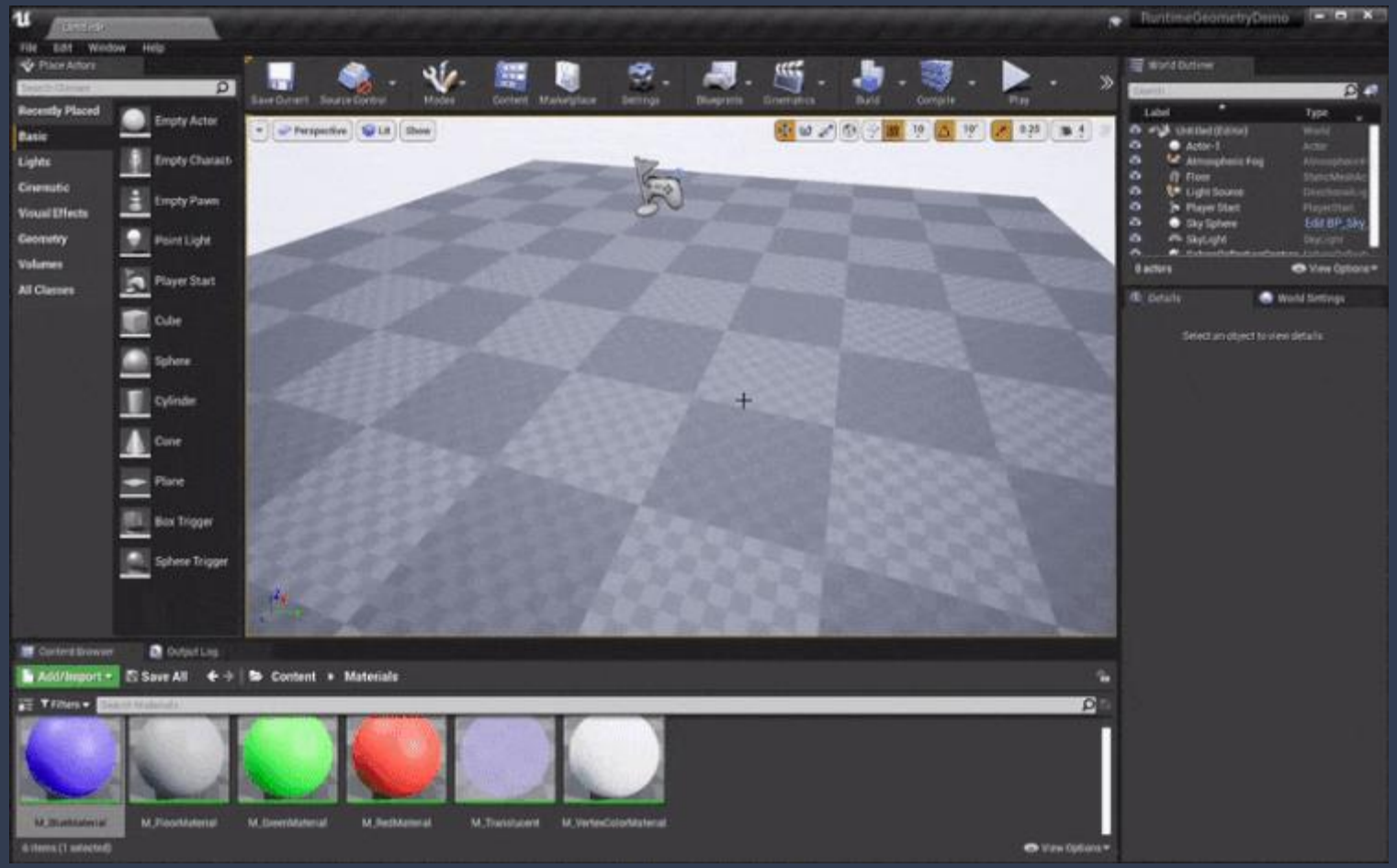
**Import Options**

| | |
|---|---|
| Import Path | SampleOBJFiles\Bunny.obj |
| Reverse Orientation | ☑ |
| Center Pivot | ☑ |
| Import Scale | 10.0 |

**Spatial Query Options**

| | |
|---|---|
| Enable Spatial Queries | ☑ |
| Enable Inside Queries | ☑ |

# 网格生成


RuntimeGeometryDemo
File Edit Window Help
Place Actors
Recently Placed
Basic
Lights
Cinematic
Visual Effects
Geometry
Volumes
All Classes
Empty Actor
Empty Pawn
Point Light
Player Start
Cube
Sphere
Cylinder
Cone
Plane
Box Trigger
Sphere Trigger
Save Current
Source Control
Modes
Content
Marketplace
Settings
Blueprints
Cinematics
Build
Compile
Play
Perspective
World Outliner
Label
Type
Details
World Settings
Select an object to view details
Content Browser
Output Log
Add/Import
Save All
Content
Materials
M_FloorMaterial
M_GreenMaterial
M_RedMaterial
M_Translucent
M_VertexColorMaterial

# 网格生成

- “导入网格”选项上的一个注释--导入路径可以是完整的C:\样式的路径，也可以是相对于项目内容文件夹的路径。我已经在项目中包含了Bunny. obj网格。此网格将包含在打包的生成中，因为我已将SampleOBJFiles文件夹添加到“项目-打包”中的设置中的“要复制的其他非资产目录”的数组中。

- 通过这些步骤，我们现在已拥有完全动态生成和导入网格的能力。如果打开编辑器，你可以在“放置Actor”面板（通过搜索框是最简单的）中找到“Dynamic SMCActor”以及PMC和SDMC actor，将任意一个拖入场景，然后更改“详细信息视图”中的参数，网格将更新，如上方的动图所示。

## 蓝图API

- ADynamicMeshBaseActor的最后一部分是一组用于执行网格导入/复制、空间查询、布尔运算、简化的方法。这些都是标记为蓝图可调用，我将在下面更详细的解释他们。但是，让我们看看其中之一的代码，只是为了看看它是什么样的：

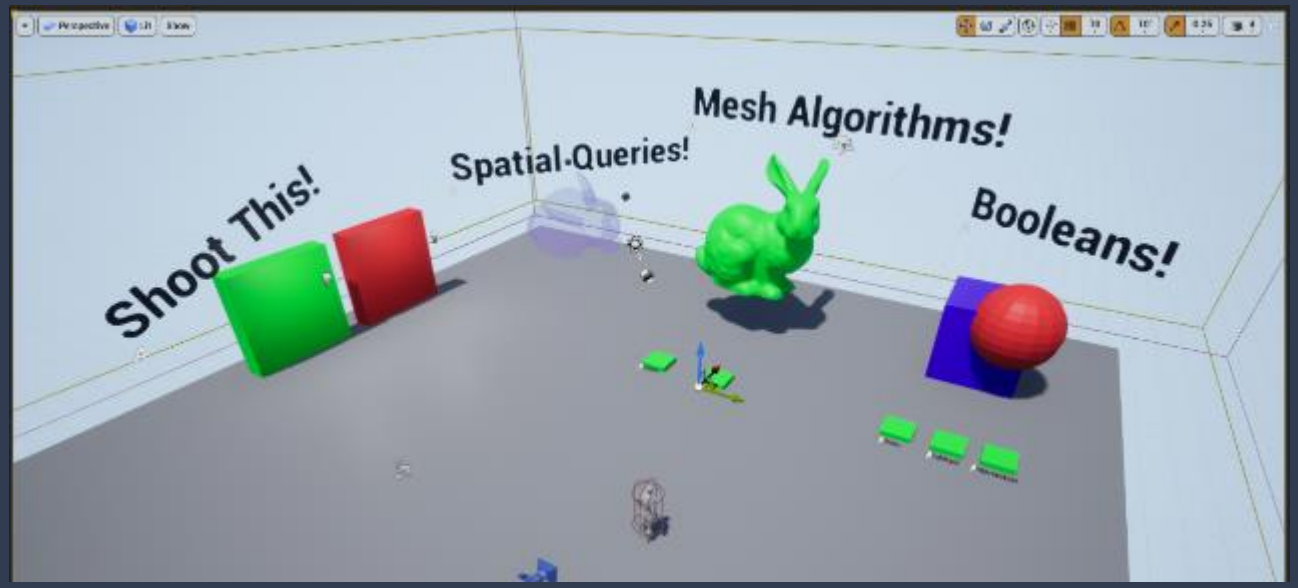
```
void ADynamicMeshBaseActor::CopyFromMesh(ADynamicMeshBaseActor* OtherMesh, bool
bRecomputeNormals)
{
    // the part where we generate a new mesh
    FDynamicMesh3 TmpMesh;

    OtherMesh->GetMeshCopy(TmpMesh);

    // apply our normals setting
    if (bRecomputeNormals)
    {
        RecomputeNormals(TmpMesh);
    }

    // update the mesh
    EditMesh([&](FDynamicMesh3& MeshToUpdate)
    {
        MeshToUpdate = MoveTemp(TmpMesh);
    });
```

- 这是一个很简单的函数，但关键是，你基本上可以剪切和粘贴此函数，重命名它，并在“the part where we generate a new mesh”插入任何几何处理代码，你将有权在蓝图中访问该操作。进行网格编辑的其他蓝图API函数正是这样做的。一个有用的练习是添加Remesh()函数，通过从Command-Line Geometry Processing Tutorial中将相关的网格重建的代码拷贝黏贴过来。

# 项目

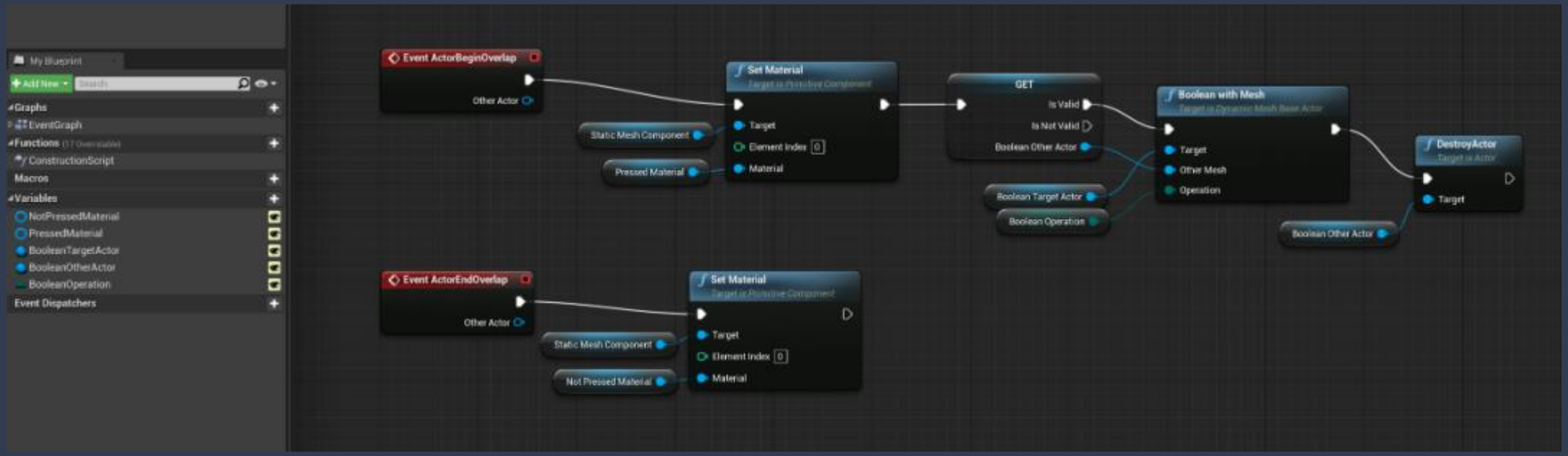
- 然后RuntimeGeometryDemo项目有一个地图，AA_RuntimeGeometryTest有4个“示例”。每个区域根据我添加到ADynamicMeshBaseActor的BP API在运行时对几何体执行一些不同的操作。从右到左，我们有一个布尔运算，网格简化/“固化”，空间查询，以及另一个布尔运算演示，在那里你可以通过射击“嚼碎”掉你面前的障碍物！

- 本教程顶部的嵌入式视频快速展示了上述功能演示。我们将在下面详细介绍它们。



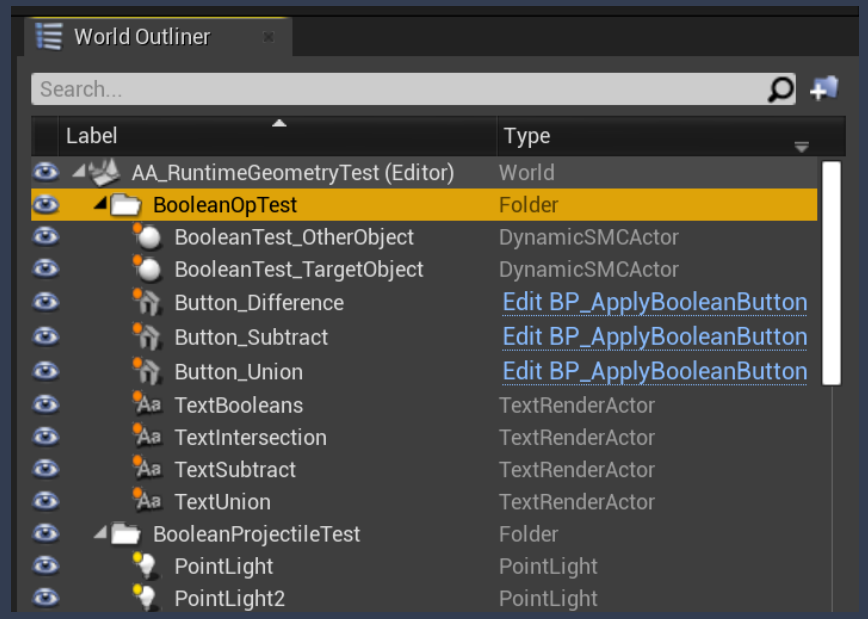
# 布尔运算!

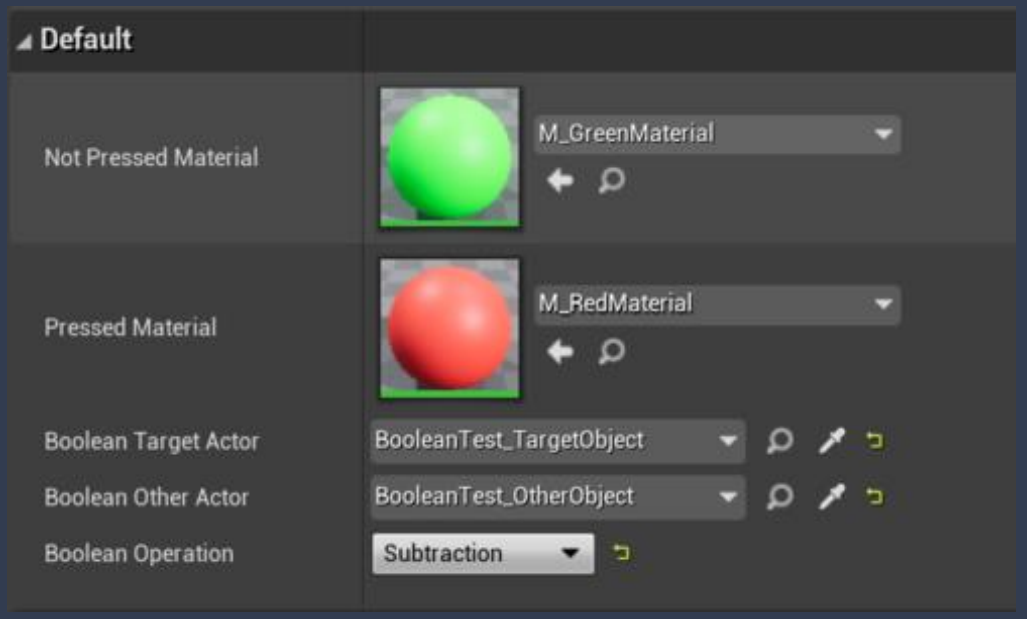
- 在第一个演示区域中，有两个动态网格Actor（本例中为SMC，但并不必须是SMC）。红色球体正在做着transformation的动画，当你踩到地面上标有“求合集、求差集”和“求交集”的三个按钮之一时，布尔操作将应用于这两个对象，并且球体将被删除。负责控制的Actor BP_ApplyBooleanButton，代码如下所示。基本上，当你踩到不同的绿色框，它将改变材质，然后我们得到目标和另一个Actor，并调用ADynamicMeshBaseActor::BooleanwithMesh（目标，另一个）。然后第二个Actor将会被销毁。就辣么简单!



# 布尔运算!

- 每个演示的对象被分组到一个文件夹中，如下图所示。下面另一张是上述Button_XYZ Actor蓝图的“详细信息视图”属性。在这里，可以为每个Button蓝图实例配置各种参数。基本上，你将“目标”和“其他”参与者设置为当前关卡的两个网格参与者，然后选择一个布尔运算操作。此外，你还可以设置按钮"按下"和"抬起"的材质。SM_ButtonBox静态网格在蓝图配置里仅注册了与Pawn重叠的事件，因此上图的ActorBeginOverlap仅在玩家角色与按钮重合时触发，然后调用布尔运算。



# 网格算法!

- 一个小的BP提示，上面使用的“验证获取”，只有在BooleanOtherActor"Is Valid"时，才能通过右键单击正常参数获取节点并选择上下文菜单底部的“转换为已验证获取”来快速创建。我只在浪费了无数分钟去写显式的IsValid分支才得知这一点。

- 在下一演示，你可以反复跳转或步行“简化”按钮，以每次将绿色兔子网格简化50%，或踩上“细化”以运行基于快速网格缠绕编号的加面。这两个本质上是剪切和粘贴BP_ApplyBooleanButton，但他们不会创建第二个Actor。下面我放大了BP_ApplySimplifyButton里相关部分。

## 空间查询!

- 第三个区域不是真正的互动，虽然如果你进入那里，并跳跃在正确的时间，你可以敲周围的飞行球体。有两个对象“附加”在半透明兔子。也是不是编辑网格，他们只是正常的静态网格演员，BP_MagnaSphere和BP_RotatoSphere。他们的蓝图是更复杂的，我已经显示了BP_RotatoSphere下。基本上，这一个只是移动在一个圆圈（“然后 0”关闭序列节点），移动一个小球体子组件到兔子网格曲面上的最近点（“然后 1”分支），然后根据StaticMesh的位置是兔子内部还是外部（“然后 2”分支）更改其颜色。

- 后两个步骤调用目标ADynamicMeshBaseActor上的ContainsPoint() 和DistanceToPoint()函数。这些只是用于如果bEnableSpatialQueies和bEnableInsideQueries属性分别为true，查询为SourceMesh自动构建的AABBTree和FastWindingTree的实用程序函数。否则，他们只会返回false。如果网格每帧更改，则构建这些数据结构可能非常昂贵，除非需要它们，否则应禁用这些结构。

## 空间查询!

- 还有一个ADynamicMeshBaseActor::IntersectRay()函数暴露在蓝图中，在任何示例中都未使用。你可能会发现此功能很有用，因为运行时生成的网格不一定受到LineTraces 的命中。对于PMC和SMC，它需要运行时物理烘焙，chaos并不完全支持，而且通常有些昂贵，SDMC完全不支持它。此外::IntersectRay()是基于双精度DynamicMesh3/AABBTree实现的，这对于巨大的网格是很有帮助的。（在编辑器中的许多建模模式操作期间，我们重建AABBTree，在"编辑工具"上下文中，它并不非常昂贵。）

- 其BP_MagnaSphere相似，只是它使用最近的点向球体添加脉冲，基本上"吸引"到兔子表面。这有时会飞向世界，所以你可能并不总是看到它。

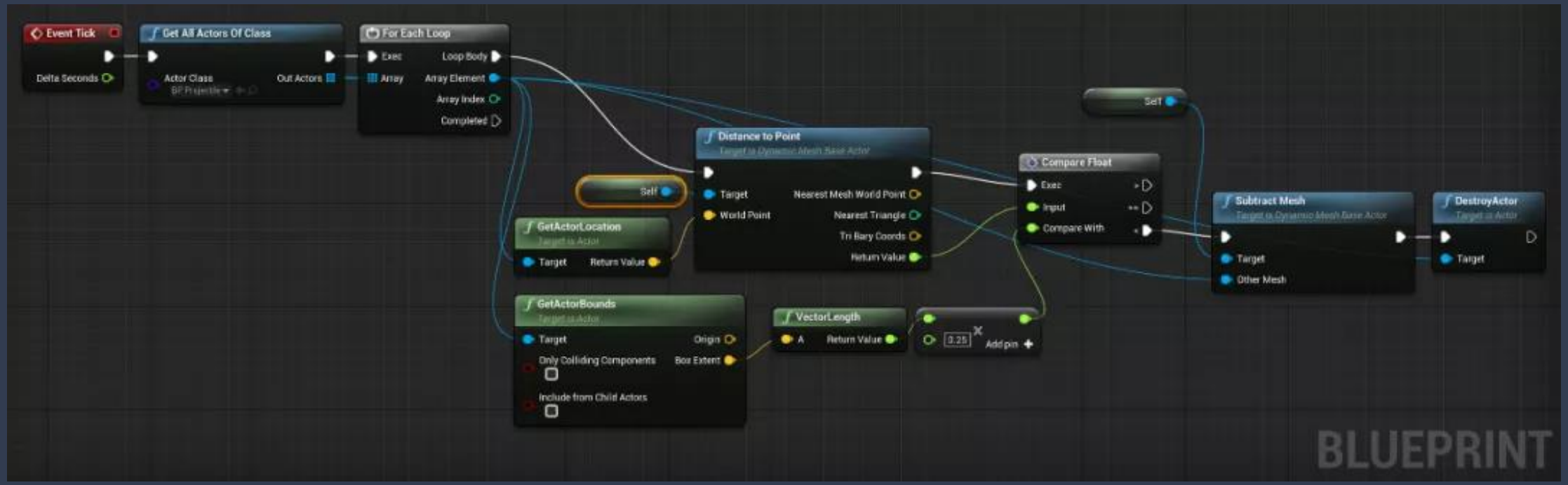
## 开枪!

- 在最后一示例，你可以左键单击红色和绿色墙壁的火球，当球体撞到墙壁时，这些球体将被减去。这里没有涉及物理，下面的蓝图使用ADynamicMeshBaseActor::DistanceToPoint()查询检测命中，该查询位于BP_Projectile的中心和墙面网格之间。将距离与项目边界框半径的一小部分 （0.25） 进行比较——这是下面比较浮点节点的输入。如果它在范围内，则从墙上减去弹丸网格，然后销毁。红墙是SDMC，绿色墙是PMC，弹球体是SDMC，但这里的点在于，在这种情况下，用哪种组件真的并不重要。

- 如上所述，每次求差集运算后，必须更新网格并重新计算AABBTree，以便能正确计算某个点到网格的距离。这听起来可能很慢，但即使我拼命对墙射击，它通常仍旧能以90-100fps在PIE中运行（在控制台中运行“stat fps”以查看帧速率）。这运行速度令人惊讶（我很惊讶！）升高墙壁或弹丸上的细化级别会对性能产生明显影响，很容易最终导致某个时候发生卡顿。需要注意的是，如果你只是点击“播放”并靠近墙体上，第一示例区域中的不断缩放的红色球体会在每帧中不断被更新，这会减慢速度（请参阅上面对PMC/SMC 讨论，红色球体是SMC）。如果禁用红色球体上的“在 Tick 上重新生成（Regenerate on Tick)”，或者先跳到其中一个布尔运算按钮（触发布尔运算后，该红球会被销毁），你就会注意到你的射击变得更流畅。

# 开枪!

- 不建议经常使用“Get All Actors of Class”（尤其在频繁被调用的函数中，比如这里的Tick）……但它的确让是个非常有用并且方便的功能!



# 开枪!

- 使用物理解算做这类事情可能会带来问题，因为通常物理碰撞检测需要事先生成“简单的碰撞”几何体，即箱形、球体、胶囊或凸体。将被炸的墙的复杂网格分解成这些形状是非常困难（man去声）的。当然也可以使用“复杂碰撞”，但在这种情况下，碰撞测试比单个点的距离查询要费得多。至于使用射线求交，事实上，我最初的确是这样做的，但这意味着球（子弹）可以很容易地“通过”小孔。而最终采用的距离阈值让结果看上去更好（降低阈值可以让每个球造成更多的伤害）。最终，就像下图一样，这个射击演示玩起来就很像那么回事儿了……

## 开枪!

- 请注意，我们也可以测试两个网格之间的精确重叠。TMeshAABBTree具有一个TestIntersection()函数，该函数需要第二个TMeshAABBTree（和可选的transform参数），这很容易通过BP暴露，就像DistanceToPoint()一样。然而，在这种情况下，这意味着球可能没机会在穿透墙壁之前触发求差集运算（这将是复杂碰撞的情况）。性能上此函数比简单的距离查询费得多。

- 最后一点，BP_Projectile的由ARuntimeGeometryDemoCharacter的Fire()函数生成的。这当然也可以在BP完成。但注意，这是我在第三人称模板项目自动生成后的工程里唯一添加的代码——C++中的所有内容都由RuntimeGeometryutils插件完成。

## 碰撞呢?

- 你会注意到，演示中所有的运行时生成的网格都没有碰撞。正如我上面描述的，布尔枪（第四个演示区域）不使用碰撞系统，即使对于实现射线追踪也没用使用碰撞系统。物理烘焙与渲染烘焙是分开的，要支持运行时物理烘焙是相当复杂的。UProceduralMesh组件确实支持运行时物理烘焙，但目前仅限于PhysX。SMC的情况更加不明朗，因为才刚刚支持运行时生成没多久，因此不清楚运行时物理烘焙是否有效（应该有与PhysX类似前提条件）。SDMC一点也不支持物理，因为它用于在编辑过程中快速可视化，而物理烘焙的成本太昂贵!

- 然而，即使我们真需要物理解算，我们仍旧面临一个复杂的问题，因为正如我上面提到的，“简单碰撞体”是物理模拟（比如物体移动）的必备前提，而且其只允许使用球体，胶囊，盒子和凸包。因此，想要运行时物理模拟一个复杂的模型我们必须先将其表示为一组简单碰撞体的组合。想对任意复杂模型自动生成这些简单碰撞体的组合在当前的技术水平下并不可行——想想如何把上图中被兔子子弹砸烂的墙分成若干盒子和球体的组合！至于另一个选项——“复杂碰撞”，它仅适用于静态对象（即可以与之碰撞，但无法物理模拟）。鉴于烘焙和测试碰撞的成本都很高，所以可能只适合在一下小的测试或者原型场景中使用，而不是在一个游戏或者应用中大量依赖于它。

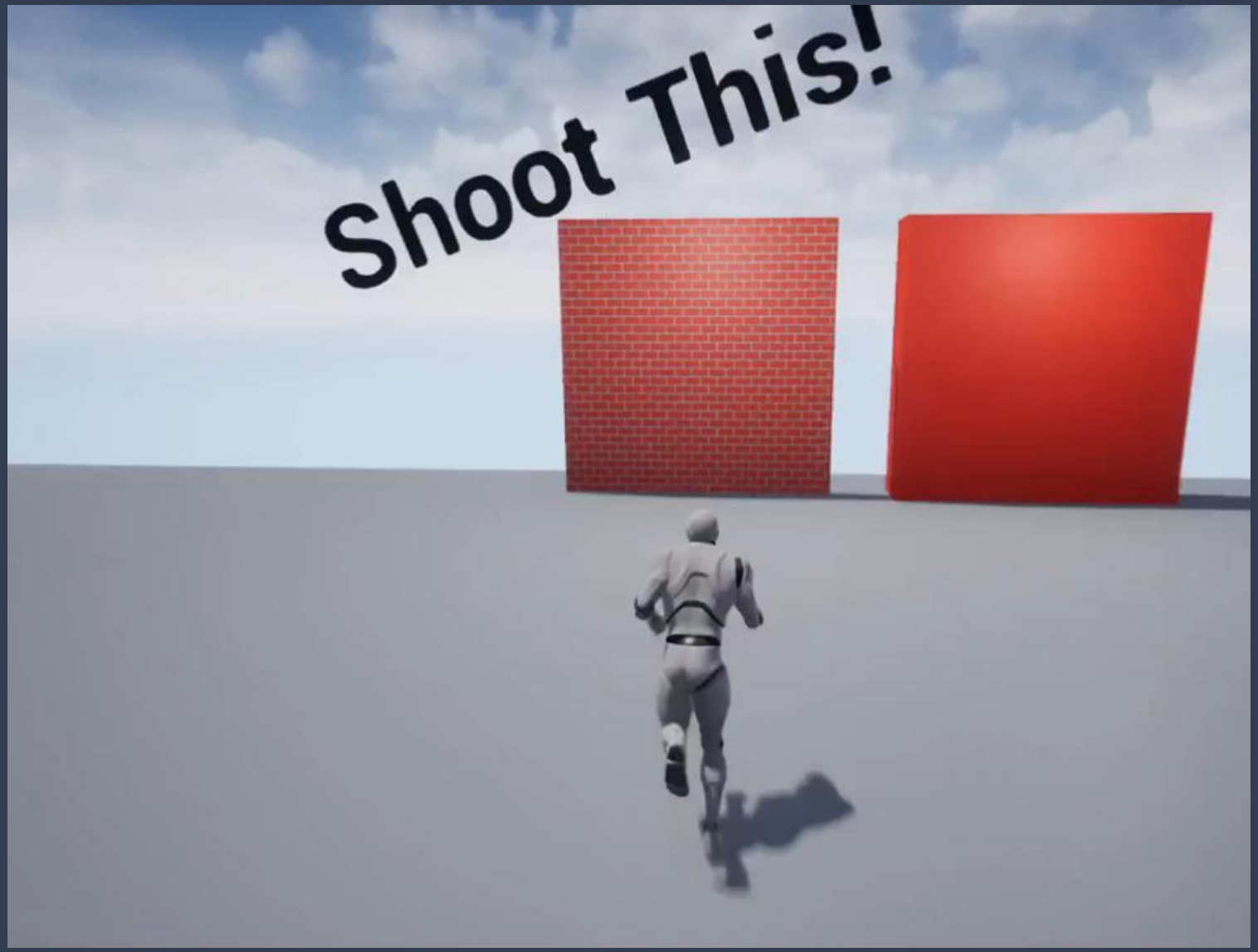
## 碰撞呢?

- 毫无疑问，这对于任何依赖于运行时生成几何的游戏或应用程序都很棘手。正如我上面展示的那样，即使没有完整的物理系统的情况下，你仍可以实现一些物理效果。并且将来完全有可能让UPrimitiveComponent的LineTrace和重叠测试能够正确支持FDynamicMeshAABBTree，这将让许多事“可行”（但物理模拟除外）。也许是未来文章的一个好主题!

## 后期的重大更新!

- 自UE4.26预览版5起，默认物理系统已切换回二进制版本的PhysX。如果你自己从源代码来构建，你仍然可以启用Chaos，并尝试未来即将推出的所有新功能。然而，这一改变导致的一个结果是UProceduralMesh组件的运行时物理烘焙再次可用。因此，我已经添加了一个碰撞模式的属性到ADynamicMeshBaseActor，包含“Complex as Simple（将复杂模型转换为简单碰撞体）”与“Complex as Simple Async（复杂模型异步转换为简单碰撞体）”两个选项。这些选项目前只会对ADynamicPMCActor产生影响，而对SMC或SDMC没有影响。它可能也可以对SMC起作用，我自己还没有试过（也许你能自己搞定，然后给我发一个PullRequest！）

# 后期的重大更新!



## 后期的重大更新!

- 上面的视频展示了如果将BP_ShootableBlock_PMC对象换为使用这些新模式之一后的情况（我也将此视频中的材质改为体积过程砖纹理，这就是为什么墙被破坏后的部分仍看起来像砖块）。由于烘培并不是每帧发生， 所以实际体验上是相当快速， 至少我个人没有注意到任何真正的性能卡顿。令人振奋!

- 有关“异步生成简单碰撞体”选项的简短说明。这意味着烘焙是在后台线程上完成的，因此它不会阻止游戏线程。这提高了性能，但代价是需要更新的碰撞几何体不一定立即可用。因此，如果你有快速变化的几何体或快速移动的对象，它们可能会更容易地导致在碰撞完成更新之前“卡”在网格内部的情况发生。

## 你应该开发自己的组件吗?

- 当更深入地研究运行时几何体生成时，经常会出现这个问题。我最常看到的情况是你使用一个具有自己网格格式的C++库来生成你自己的网格模型。你希望在虚幻引擎中能显示这个格式的网格模型。所以正如我们在本教程中所做的那样，你必须将自己的数据塞到PMC或SMC中，才能在屏幕上显示它。将其变成一个引擎可以“消化”的数据格式（无论是PMC Sections，FMeshDescription，或现在SDMC中的FDynamicMesh3）前你可能需要对你的网格模型格式进行一次或多次复制/转换 。

- 但是，如果你深入到PMC代码中，你会发现PMC和SceneProxy并不复杂。不用花很长时间，你就能发现如何绕开“中间商”，实现一个你自己的网格组件，这个组件能够直接从你自己的网格格式的实例中读取数据来初始化渲染缓冲区。事实上，SDMC基本就是这么干的，当然还有一种可选择的网格格式是用FSimpleDynamicMesh3。

## 你应该开发自己的组件吗?

- 那么你应该这么做吗？我的观点是这取决于你想投入多少时间来保存一些网格副本。如果你有一支技术娴熟的工程师队伍，那么这不是一项巨大的工程，但随着引擎的发展，你必须不断地保持组件的更新。如果你是一支小型团队或独立团队，那么这种程度的性能优化可能不值得付出努力。我当然想分析网格复制/转换的成本，但我们的实验发现这不是瓶颈。将新的渲染缓冲区上传到GPU仍然是最费时的地方，这不会随着你采用的组件类型而改变。

- 如果你像本教程中那样组织“网格体系结构”，则不一定有什么影响——如果你不依赖于某个特定组件的功能，你可以根据需要换成新的组件类型。

## 你应该开发自己的组件吗?

- 增编：khammassi ayoub已经写了一系列非常详细的文章来介绍如何在UE里创建自己的网格组件[567]。然而需要注意的一个小地方是在他的教程中，大部分的工作是关于如何自定义顶点着色器，但如果你只是想做一个“类似PMC但用自己的网格格式，以避免复制”的网格组件，自定义着色器是没有必要的。尤其当你只是渲染“正常”网格，你不需要实现自己的顶点工厂/等。但这些文章仍旧非常值得一读因为其中涵盖了渲染端组件/代理基础结构的所有关键部分。

# 总结

- 本教程已近尾声。本篇中我重点介绍了布尔运算，因为它们不需要任何额外参数控制。然而，在MeshModelingToolset插件中的几何处理和建模操作器模块具有像拉伸、偏移、弯曲/扭曲/削减等空间变形器操作，这些操作可以很容易地添加到类似ADynamicMeshBaseActor的类中并在游戏中与UMG或者游戏内的特定操作进行交互。

- 虽然你可以直接在你的项目中使用ADynamicMeshBaseActor和RuntimeGeometryutils插件，但我觉这些更多的应该被视作一种指南，来帮助你建立自己的版本。如果你正在研发涉及存储运行时创建的内容的游戏或应用，我鼓励你花一些时间思考如何组织你的数据。在此演示中将源网格存储在Actor使得我后续的实现非常方便，但如果我构建的是真实内容，我就会将这些源网格的所有权从Actor转移到一些更集中的位置，例如UGameInstance子系统。你可能觉得这看起来像“可以放在稍后再去做的事儿”，但如果你基于当前的设计开发了很多蓝图或游戏功能，那之后的重构会显得非常繁琐（我最初将ADynamicMeshBaseActor放在Game目录中，后来为了能将.h/.cpp文件正确的移动到插件目录中而不破坏已有的功能，我花了一个下午来学习UE的Redirector……）

# 总结

- 也并不是一定需要将PMC/SMC/SDMC分成不同的Actor。因为我想在演示中对其进行比较，所以这样把它们分开比较方便。但如果有一个Actor基类可以通过一个enum属性动态的生成不同的网格组件也是极好的。这可以使蓝图编程变得更加轻松。因为现在，如果你为一个Actor子类制作一个BP，并且想要将其切换到其他子类，你必须跳到好几层外面来更改其父类，并且无法在多种Actor之间共享功能（这就是为什么教程中有两个可被子弹破坏的墙的BP类，一个基于PMC，一个基于SDMC）。
- 最后，关于我前面提到的目前该演示无法在OSX上运行。这是因为USimpleDynamicMesh组件——它是MeshModelingToolset插件的一部分，该插件依赖于一些仅能在Windows运行的第三方模块。通过一些“#ifdef”应该可以让包含SDMC的建模组件（ModelingComponents）模块在OSX上运行起来，但这需要重新编译引擎。更直接的解决方案是删除ADynamicSDMCActor.h/.cpp和RuntimeGeometryUtils.build.cs文件中对“建模组件（ModelingComponents）”的引用，并且仅在演示工程中使用SMC或PMC的Actor。我已经在Windows上验证过，如果这样做，一切都仍然能够编译，这意味着它应该也能在OSX上工作，但我自己还没在OSX上进行测试。（请注意，这样的改动意味着大部分示例项目在不修改的前提下无法正确运行。）