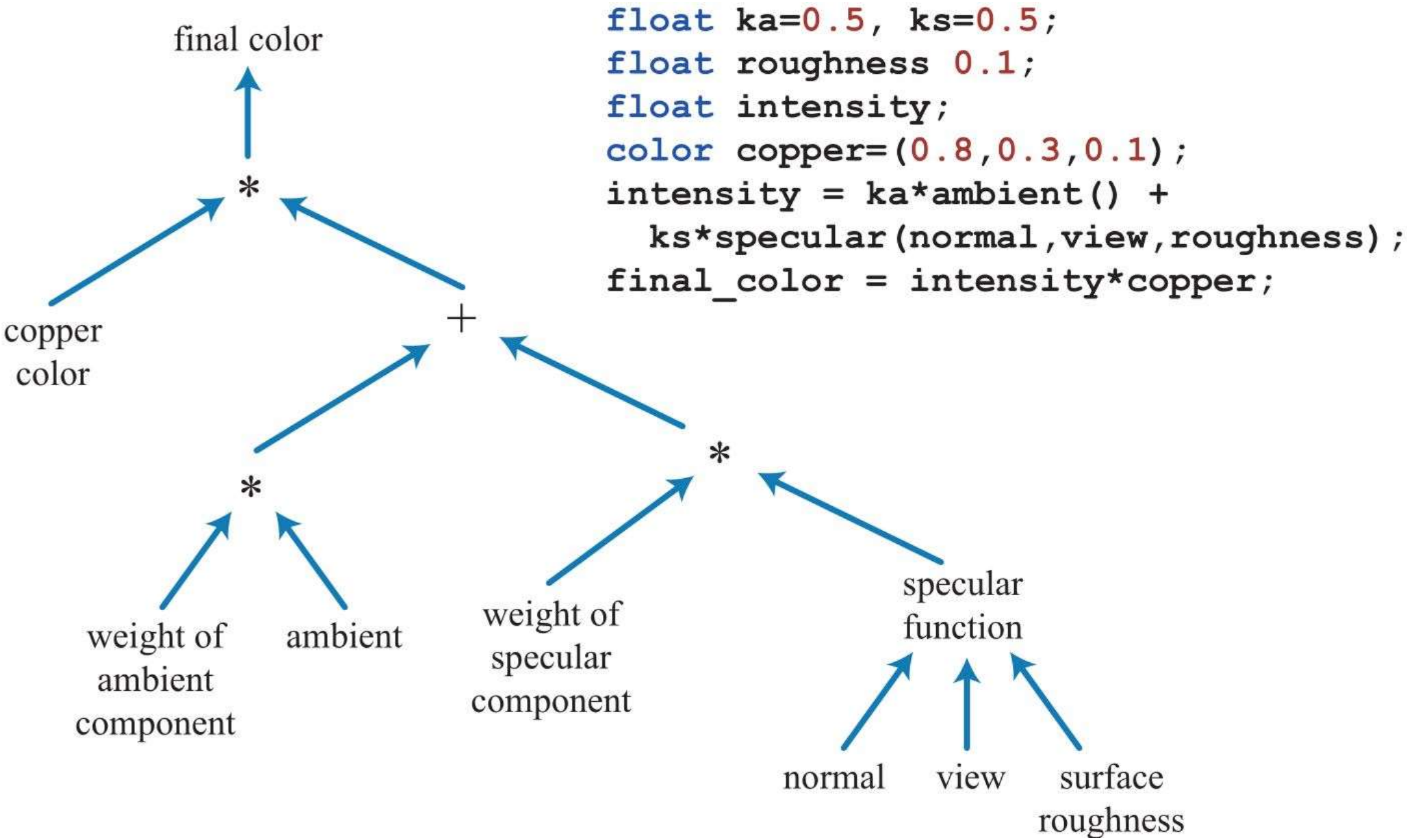
3.4 可编程着色和及其API的演变

可编程着色框架的想法可以追溯到1984年Cook的《shade trees》[287]。图3.4显示了一个简单的着色器及其相应的着色树。RenderMan着色语言[63, 1804]是在1980年代后期从这个想法发展而来的。它今天仍然用于电影制作渲染，以及其他不断发展的规范，例如开放着色语言(OSL)项目[608]。




```
float ka=0.5, ks=0.5;
float roughness 0.1;
float intensity;
color copper=(0.8,0.3,0.1);
intensity = ka*ambient() +
  ks*specular(normal,view,roughness);
final_color = intensity*copper;
```

图3.4. 一个简单的铜材质着色器的着色树，及其相应的着色器语言程序。（在Cook [287] 之后。）

消费级图形硬件于1996年10月1日由3dfx In-teractive首次成功推出。有关今年的时间表，请参见图3.5。他们的 Voodoo显卡能够以高品质和高性能渲染游戏Quake，因此很快被采用。该硬件自始至终都实现了一个固定功能的流水线。在GPU原生支持可编程着色器之前，曾多次尝试通过多个渲染通

道实时实现可编程着色操作。Quake III:Arena脚本语言是1999年该领域第一个广泛的商业成功。正如本章开头提到的，NVIDIA的GeForce256是第一个被称为GPU的硬件，但它不可编程。但是，它是可配置的。

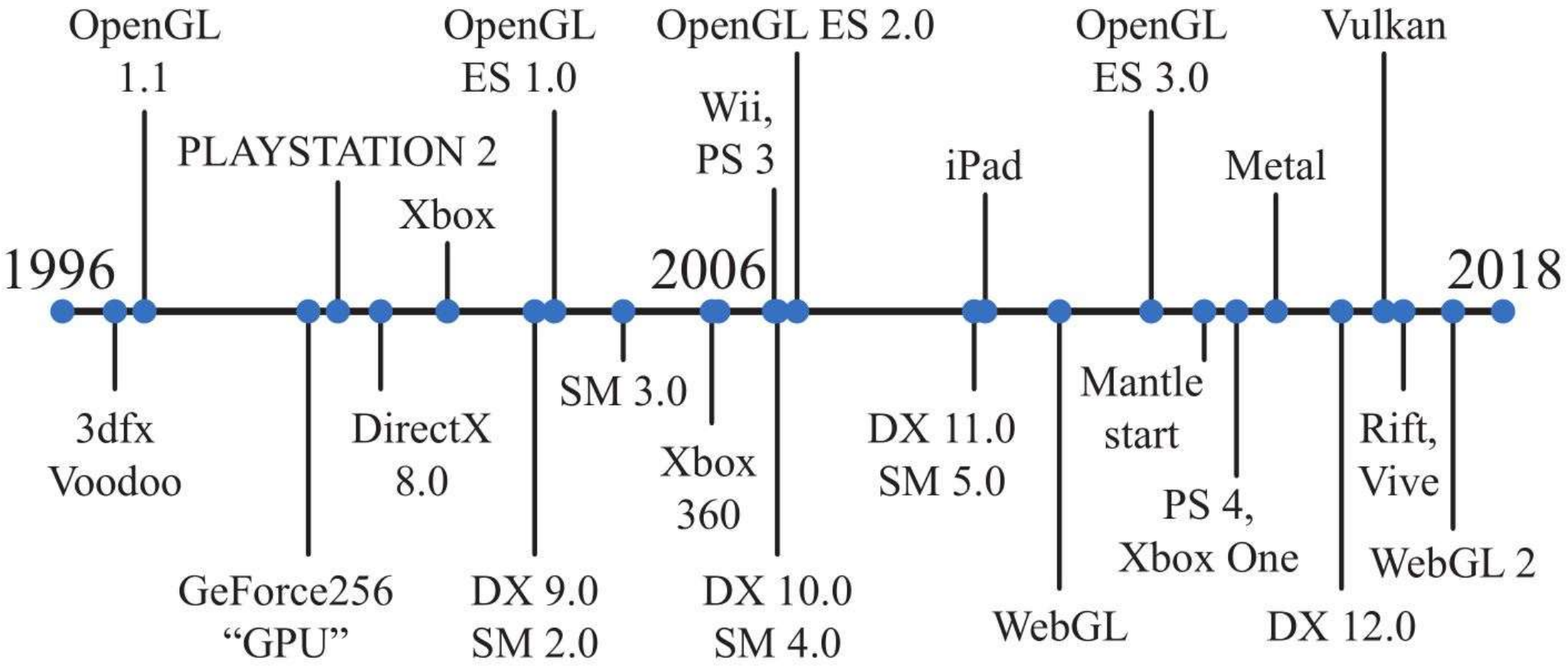


图3.5. 一些API和图形硬件发布的时间表。

2001年初，NVIDIA的GeForce 3是第一个支持可编程顶点着色器的GPU[1049]，通过DirectX8.0和对OpenGL的扩展来开放接口。这些着色器是用类似汇编的语言编程的，驱动程序可以将其即时转换为微代码。像素着色器也包含在DirectX 8.0中，但像素着色器缺乏实际的可编程性——支持的有限“程序”被驱动程序转换为纹理混合状态，驱动程序又将硬件“寄存组合器”连接在一起。这些“程序”不仅长度有限（12条指令或更少），而且缺乏重要的功能。Peercy等人从他们对RenderMan的研究[1363]中发现，依赖纹理读取和浮点数据对真正的可编程性至关重要。

此时的着色器不允许流控制（分支），因此必须通过计算两个条件分支并在结果之间选择或插值来模拟条件。DirectX定义了着色器模型 (SM) 的概念，以区分具有不同着色器功能的硬件。2002年，DirectX9.0发布，包括Shader Model 2.0，它具有真正可编程的顶点和像素着色器。使用各种扩展在OpenGL下也公开了类似的功能。添加了对任意依赖纹理读取和16位浮点值存储的支持，最终完成了Peercy等人确定的一组要求。对着色器资源（例如指令、纹理和寄存器的限制有所增加，因此着色器能够实现更复杂的效果。还增加了对流控制的支持。着色器的长度和复杂性不断增加，使得汇编编程模型变得越来越繁琐。幸运的是，DirectX 9.0还包含 HLSL。这种着色语言是由Microsoft与NVIDIA合作开发的。大约在同一时间，OpenGL ARB（架构审查委员会）发布了GLSL，这是一种与OpenGL非常相似的语言 [885]。这些语言深受 C编程语言的语法和设计理念的影响，并包含来自RenderMan着色语言的元素。

Shader Model 3.0于2004年推出，增加了动态流控制，使着色器更加强大。它还将可选功能变成了要求，进一步增加了资源限制，并增加了对顶点着色器中纹理读取的有限支持。2005年末（微软的Xbox 360）和2006年末（索尼电脑娱乐的PLAYSTATION 3系统）推出新一代游戏机时，都配备了Shader Model 3.0级别的GPU。任天堂的Wii游戏机是最后一批著名的固定功能GPU之一，最初于 2006 年底发货。纯粹的固定功能管道在在之后已不复存在。着色器语言已经发展到可以使用各种工具来创建和管理它们的地步。图3.6显示了使用Cook着色树概念的此类工具的屏幕截图。

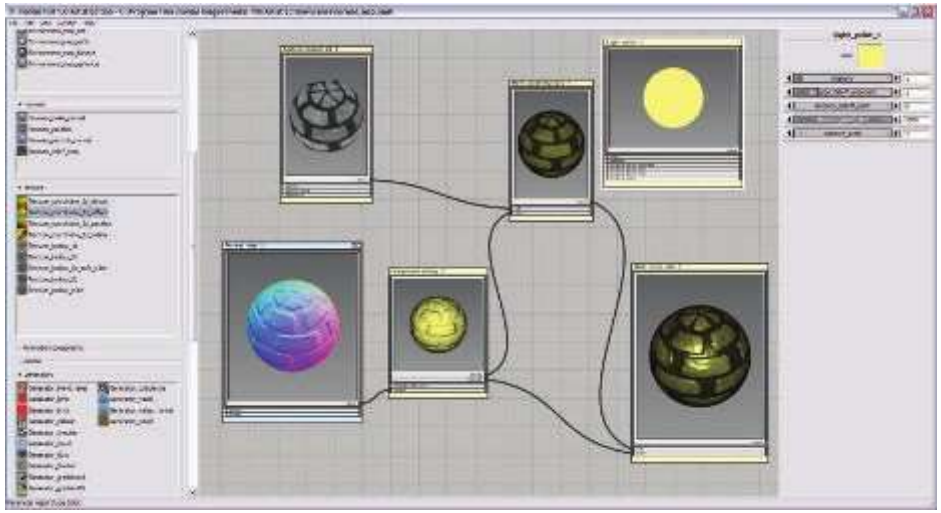


图3.6. 用于着色器设计的视觉着色器图形系统。各种操作都封装在功能框中，在左侧可选择。选中后，每个功能框都有可调参数，如右图所示。每个功能框的输入和输出相互链接以形成最终结果，显示在中心框的右下方。（截图来自“mental mill”，mental images inc。）

可编程性的下一个重要步骤也在2006年底附近发生。包含在DirectX 10.0[175]中的着色器模型4.0引入了几个主要功能，例如几何着色器和流输出。Shader Model 4.0包括一个适用于所有着色器（顶点、像素和几何）的统一编程模型，即前面描述的统一着色器设计。进一步增加了资源限制，并添加了对整数数据类型（包括按位运算）的支持。OpenGL 3.3中GLSL 3.30的引入提供了类似的着色器模型。

2009年DirectX 11和Shader Model 5.0发布，添加了曲面细分阶段着色器和计算着色器，也称为DirectCompute。该版本还专注于更有效地支持CPU多处理，这是第18.5节中讨论的主题。OpenGL在4.0版中添加了曲面细分，在4.3版中添加了计算着色器。DirectX和OpenGL的发展方式不同。两者都设置了特定版本发布所需的一定级别的硬件支持。Microsoft控制DirectX API，因此直接与AMD、NVIDIA和Intel等独立硬件供应商(IHV)以及游戏开发商和计算机辅助设计软件公司合作，以确定要公开的功能。OpenGL由硬件和软件供应商联盟开发，由非营利组织Khronos Group管理。由于涉及的公司数量众多，API功能通常在DirectX中引入后的某个时间出现在OpenGL的发行版中。但是，OpenGL允许特定于供应商或更通用的扩展，允许在发布正式支持之前使用最新的GPU功能。

API的下一个重大变化是AMD于2013年推出的Mantle API。与视频游戏开发商DICE合作开发，Mantle的想法是剥离大部分图形驱动程序的开销，并将此控制直接交给开发人员。除了这种重构之外，还进一步支持了有效的CPU多处理。这类新的API侧重于大大减少CPU在驱动程序中花费的时间，以及更高效的CPU多处理器支持（第18章）。 Mantle中首创的想法被微软采纳并于2015年作为DirectX 12发布。请注意，DirectX 12并不专注于展示新的GPU功能——DirectX 11.3展示了相同的硬件功能。这两个API都可用于将图形发送到虚拟现实系统，例如Oculus Rift和HTC Vive。然而，DirectX 12是对API的彻底重新设计，更好地映射到现代GPU架构。低开销驱动程序对于CPU驱动程序成本导致瓶颈的应用程序很有用，或者使用更多的CPU处理器来处理图形可以提高性能[946]。从早期的API移植可能很困难，而且一个不成熟的实现会导致较低的性能[249, 699, 1438]。

Apple于2014年发布了自己的低开销API，称为Metal。Metal最初可用于iPhone 5S和iPad Air等移动设备，一年后更新的Macintosh可通过 OS X El Capitan访问。除了效率之外，降低CPU使用率还可以节省电量，这是移动设备的一个重要因素。这个API有自己的着色语言，适用于图形和GPU计算程序。

AMD将其Mantle工作捐赠给了Khronos Group，后者于2016年初发布了自己的新API，称为Vulkan。与OpenGL一样，Vulkan可在多个操作系统上运行。Vulkan使用一种称为SPIRV的新高级中间语言，它用于着色器表示和通用GPU计算。预编译着色器是可移植的，因此可以在支持所需功能的任何GPU上使用[885]。Vulkan也可用于非图形GPU计算，因为它不需要显示窗口[946]。Vulkan与其他低开销驱动程序的一个显着区别在于，它旨在与从工作站到移动设备的各种系统一起使用。

在移动设备上，标准是使用OpenGL ES。“ES”代表嵌入式系统，因为此API是为移动设备开发的。当时的标准OpenGL在其某些调用结构中相当庞大和缓慢，并且需要支持很少使用的功能。OpenGL ES 1.0于2003年发布，是OpenGL 1.3的精简版本，描述了一个固定功能的管道。虽然DirectX的发布与支持它们的图形硬件的发布同步，但为移动设备开发图形支持并没有以同样的方式进行。例如，2010年发布的第一款iPad实现了 OpenGL ES 1.1。2007年，OpenGL ES 2.0规范发布，提供可编程着色。它基于OpenGL 2.0，但没有固定功能组件，因此不向后兼容 OpenGL ES 1.1。OpenGL ES 3.0 于2012 年发布，提供了多个渲染目标、纹理压缩、变换反馈、实例化以及更广泛的纹理格式和模式以及着色器语言改进等功能。OpenGL ES 3.1添加了计算着色器，3.2添加了几何和曲面细分着色器等功能。第23章讨论更详细的移动设备架构。

OpenGL ES的一个分支是基于浏览器的 API WebGL，通过JavaScript调用。该API的第一个版本于2011年发布，可用于大多数移动设备，因为它在功能上等同于OpenGL ES 2.0。与OpenGL一样，扩展可以访问更高级的GPU功能。WebGL 2假定支持OpenGL ES 3.0。

WebGL特别适合在课堂上试验特性或使用：

- 它是跨平台的，适用于所有个人计算机和几乎所有移动设备。
- 驱动程序批准由浏览器处理。即使一个浏览器不支持特定的GPU或扩展，通常另一个浏览器会支持。
- 代码是解释性的，而不是编译性的，开发时只需要一个文本编辑器。
  - 大多数浏览器都内置了调试器，可以检查在任何网站上运行的代码。
  - 例如，可以通过将程序上传到网站或Github来部署程序。

更高级别的场景图和效果库（例如three.js [218]）可以轻松访问各种更复杂的效果的代码，例如阴影算法、后处理效果、基于物理的着色和延迟渲染。