使用 A*平滑消除3D 游戏中的画面抖动

专业：计算机科学与技术

学生：谭鹏　　指导教师：陈杰

[**摘要**] 21世纪是属于互联网、属于计算机的一个世纪，也是属于电子游戏的一个世纪。在这个世纪刚刚开始的这么短短十几年里，计算机的性能和互联网得到了飞速的发展。电子游戏也以爆炸般的速度在发展。而3D 游戏由于其对现实模拟的更为逼真、更能给予玩家一种冲击感，3D 游戏站在了游戏发展的浪头上，引领者游戏的风骚。一大波的开发者正在涌入到3D 游戏的开发阵营中来。可是3D 游戏的开发过程中，很可能会遇到一些麻烦奇怪的问题，而画面抖动就是其中一种让很多开发者百思不得其解的奇怪问题。本文分析了3D 游戏中画面抖动的原因，解释了3D 游戏的一些基本概念，也为抖动提供了一种解决方案和方案的原理。旨在帮助新入3D 游戏开发的游戏开发者能够轻松的跨过这道门槛。本文消除抖动所使用的 A*算法为寻路的经典算法，本文将该算法与一些已知的算法进行组合、改造，形成了本文的算法。算法可靠性可以得到保证。同时，本文对于3D 变换的部分提供了数学理论证明，保证可靠性。

[**关键词**] 3D 游戏，平滑算法，画面抖动，A*算法

A* Smoothing for shaking in 3D Games

Major: Computer Science

Author: Peng Tan Tutor: Jie Chen

[**Abstract**] The 21st century is the century that belongs to the Internet, belongs to the computer for a century, and also belong to the video games. In the just begun so short ten years of this century, the computer's performance and the Internet has been rapidly developed. The video games are also developing at an explosive rate. The 3D games, because of its realistic simulation of more realistic, more able to give players a sense of impact, are in the on the waves of game development, and are the leader coquettish of the game. A big wave of developers are pouring into the 3D game development camp. But in the process of development of 3D games, it is likely to encounter some strange problems, and the screen shaking is one of the strange questions that a lot of developers hard to understand. This paper analyzes the causes of shaking in 3D games, explains some of the basic concepts of 3D games, and provides a solution for shaking. Designed to help the new 3D game developers can easily cross this threshold. In this paper, we use the classical algorithm of pathfinding -- A * algorithm for erasing shaking. In this paper, we combine the algorithm with some known algorithms to transform and form the algorithm. Algorithm reliability can be guaranteed. At the same time, this article provides a mathematical proof of the 3D transformation part to ensure reliability.

[**Keywords**] 3D Gaming, smoothing algorithm ,shaking, A * algorithm

目录

一、引言

1.1 背景

在计算机游戏中，模拟游戏（Sim）、射击游戏（shooter）角色扮演游戏（RPG）是较早并且具有较强代表性的计算机游戏，这些游戏的设计、产生和应用为计算机游戏整体设计水平的提升奠定的良好的基础。在这些早期的游戏设计过程中 ，游戏设计者通过让游戏玩家利用一种非结构化的方式来进行游戏体验 ，这种游戏体验往往具有较为明确的目标，例如角色扮演游戏则是以剧情的进展、而射击游戏是以关卡的演进为明确的目标，但是在明确的设计结构下游戏玩家的动作是随意的 ，即在这种游戏设计原理下游戏玩家能够进行具有探索性的游戏方式 ，这种游戏方式的存在极大的提升了计算机游戏的可玩性并且为接下来计算机游戏的设计水平提升提供了重要助力。即游戏玩家通过自主探索在实际上可以有效反应出游戏设计的基本结构 ，这种结构在提升游戏系统明确性的同时可以较为集中的反应游戏设计接下来的进步方向。[1]

随着个人电脑性能的提升、普通用户智能手机的普及，使得现如今人们所持有的设备的性能越来越高，3D 游戏所需要的额外性能很容易就可以满足。所以 3D 游戏市场变得异常火热。但是在游戏开发过程中，作为一个开发者，我们时常会遇到游戏画面抖动的情况。如果任由这种情况存在，会降低玩家的体验，是对游戏可玩性的一大损害。

1.2 意义

可玩性是如同字面一样，说的是游戏让玩家玩下去的性质。对于电子游戏这一个特殊的软件来说，可玩性是其核心内容。可玩性主要受游戏类型与游戏设计、游戏操作与响应、学习曲线和精通难度的影响。

3D 游戏相较于 2D 游戏来说，从图形学角度来讲，多了一个 z 轴，使得 3D 游戏是在空间的层次上进行展示，而 2D 是在平面的层次上进行显示。2D 游戏现实只需要将屏幕的一部分直接投射到屏幕上就可以，但是 3D 不同，由于 3D 使得游戏有远近、深浅的概念，简单的投射已经不可行。3D 游戏采用了人眼的模式，来展现游戏世界。如果游戏的

过程中，游戏画面抖动，将会使玩家感觉自身在抖动，影响玩家在游戏中的判断，降低游戏的可玩性。

由于可玩性是电子游戏的核心，消除游戏中画面抖动势在必行。采用本文的方法可以消除开发过程中出现的画面抖动，提高游戏的体验，提高游戏的可玩性，增加用户粘度，减少开发负担。

1.3 现状

在当前，3D 游戏除了一部分卡牌类游戏以外，其余的游戏基本上都需要涉及到画面的移动、人物的移动等。在这些情况下，出现画面抖动的情况非常高。应用消除抖动是势在必行的。而当前，在 3D 游戏开发过程中，遇到了画面抖动，一般采用的是漏斗平滑算法来进行消除。

而一般所采用的漏斗平滑算法在八格子寻路时效率不好，并不适用于某些类型的游戏。

1.4 其他抖动消除算法简述

漏斗平滑算法是 3D 游戏中，消除抖动时最常见的一种平滑算法。

漏斗平滑算法过程是基于边界的。每次判断下一个点是否在漏斗中，再会构建新的漏斗继续进行判断，不在则把之前在的点设为新的起点来构建漏斗，重复进行。但是这个算法是基于边界和点的。在某些情况下会效率比较低下，不适用于某些游戏。

如图 1 所示，O(t)和 U(t)构成的区域可以看作是一个单调多边形。我们用Π(s,w)来表示多边形内点 s 至点 w 的最短路径，那么 Π(s,e) 即为所求最短路径。显然，Π(s,e)只可能与多边形的凹点(例如 v)相交，不可能与凸点(例如 z)相交。因此，计算最短路径只需要考虑凹点。假设已知 Π (u,s) ＝ [u,s] 及 Π (s,w) ＝[s,v,w] ，那么漏斗就是由 Π(u,s) 以及 Π(s,w) 构成的 V 形结构，用 F_{uw} 表示，而 s (V 形底端)为漏斗的底。F_{uw}是由最初漏斗经过数次更新后生成的，最初的漏斗由起始点 s 与 O(t) 、 U(t) 的第一个凹点的连线构成的，如图中虚实线所示。

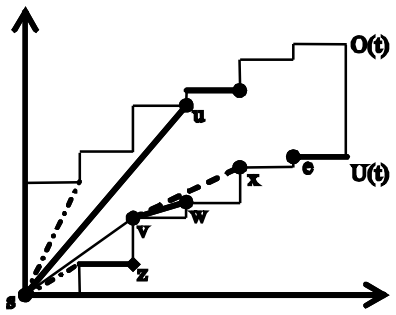


图 1

在现有漏斗 F_{uw} 的基础上，计算出漏斗底至下一凹点 x 的最短路径 Π(s,x)，那么漏斗就更新为 F_{ux} = Π(u,s)+Π(s,x)，依次类推计算后续凹点直至终点 e，即可得到 F*e，而 F*e 的下半部分即为所求的 Π(s,e)，在 F_{uw} 的基础上，计算 Π(s,x) 的方法如下。首先，如果 s 到 x 的直线不与 F_{uw} 相交，那么 Π(s,x) 就是直线 sx，否则，在 F_{ux} 中寻找一个点 v_i，使得 v_ix 与 F_{ux}相切。用公式可以表示为：

$$\mathrm{tg}(v_{i-1}v_i) \geq \mathrm{tg}(v_i x) \geq \mathrm{tg}(v_i v_{i+1})$$

其中 tg 表示线段的斜率，v_{i-1} v_i 以及 v_i v_{i+1} 为 F_{uw} 中 v_i 的前后线段。找到点 v_i 后连接 v_ix，那么 Π(s,x)=[s,…v_i,x][2]。

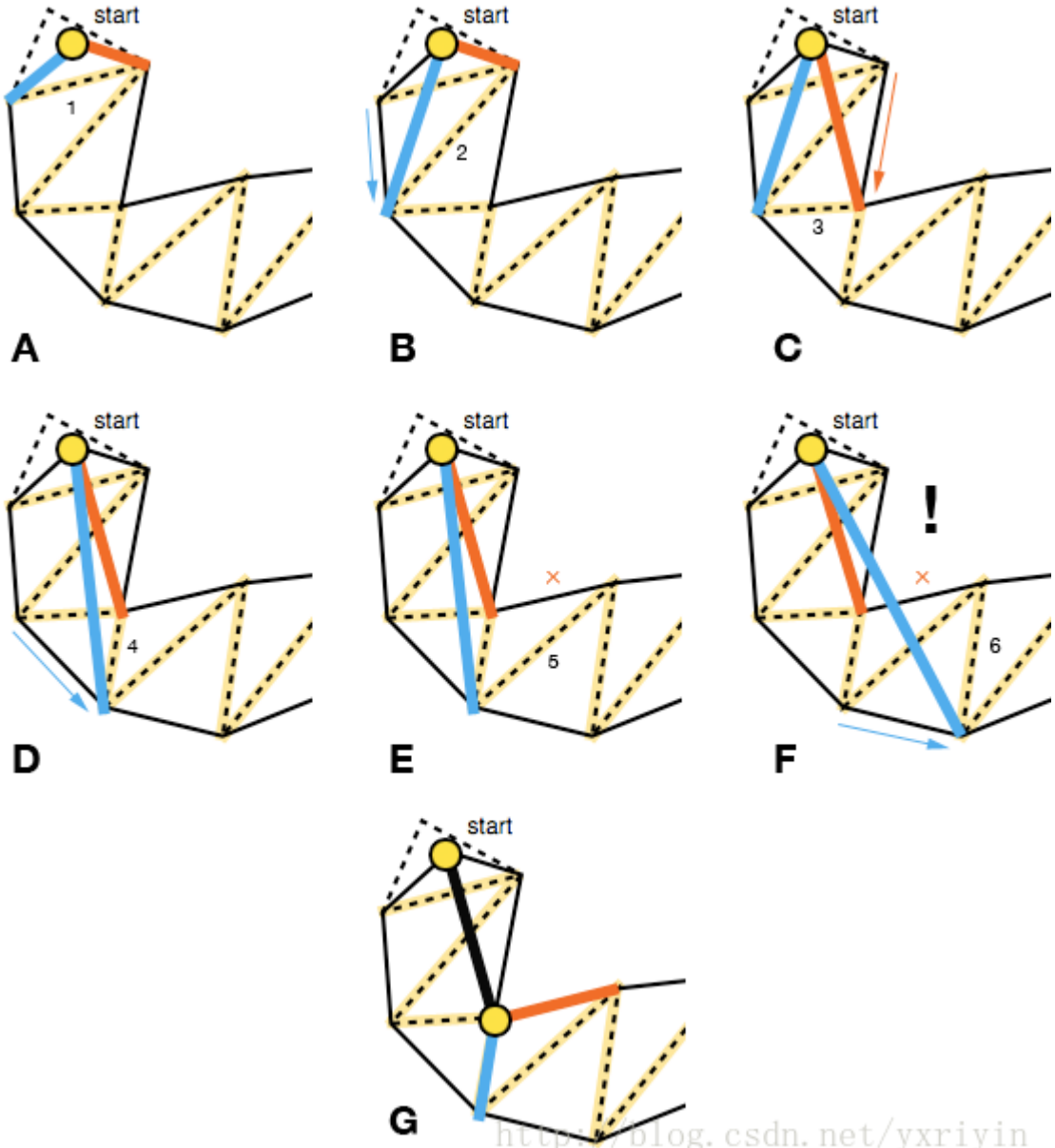


图 2

以图 2 来说明漏斗的过程：

1．起点与两边边界最开始的点连线，这两条线构成了一个漏斗。
2．分别测试两边边界上的下一个点与起点的联系是否在前面的点生成的漏斗中。如果都在，则执行 3；若一条在一条不在，则执行 4；若都不在则执行 5。
3．以新的两条线原起点构成新的漏斗，重复 2。
4．以在漏斗中的那条新线为新边界与另一边的原边界和原起点构成新的漏斗，重复 2；
5．如果都不在，则以两条线中的最短的一条中的端点为新的起点，这条线段为路径中的一段加入到结果集合中。重复 1。

图 2-A 中起点与最近的两个点组成漏斗。2-B 中下侧边界的下一点与起点连线，该线在漏斗中。图 2-C 中上边界的下一点与起点连线，连线在原漏斗中。则以这两条线为新的漏斗边。2-D 图中，下边界下一点与起点连线，线在新的漏斗中，2-E 图中，上边界下一点与起点连线，不在漏斗中，舍弃。下边界新线与原上边界线组成新的漏斗。2-F 图中下边界下一点与起点连线，不在漏斗中，舍弃，上边界下一点与起点连线，不在漏斗中，舍弃。选择原漏斗边界中最短一条的端点为新起点。如图 2-G。重新开始漏斗处理。

1.5 工作与创新点

由于漏斗平滑算法是基于边界的平滑算法，在某些游戏中，算法效率较低，并不适用，而 A*算法一般用于最短路径的查找中，不能直接应用与平滑中，于是通过对传统 A*算法的拓展，实现了一个平滑算法，用于消除 3D 游戏中出现的画面抖动。

本文所使用的算法基于 A*算法，本文中运用得数学知识，均有数学证明，保证了算法的可靠性。同时，本算法基于 A*算法，方便移植到实际项目中。

二、相关算法

2.1 A*算法

A*算法原本是一种求解最短路径直接搜索方法，是一种启发式搜索算法。

公式表示为：

$f(n) = g(n)+h(n)$

其中， f(n)是从初始状态经由状态 n 到目标状态的代价即距离，g(n)是在状态空间中从初始状态到状态 n 的实际代价即初始点到点 n 的距离，h(n)是从状态 n 到目标状态的最佳路径的估计代价即点 n 到终点的距离。

A*算法的具体过程为：

1．开启列表置为空，将起始格添加到开启列表之中。

2．寻找开启列表中当前 F 值最低的格子，即为当前格。然后将当前格加入到关闭列表。

3. 对于当前格相邻的每一个格子，如果这一格不可通过或者已经在关闭列表中，那么就略过这一格，比较下一个格子从 v；如果这一格不在开启列表中，那么就把这一格添加到开启列表中去，并且把当前格作为这一格的父节点，并记录这一格的 F,G,和 H 的值；如果这一格已经在开启列表中，则检查这一格的 G 值是否更低，如果是，就把这一格的父节点改成当前格，并且重新计算这一格的 G 和 F 值；如果你把目标格添加进了关闭列表，则说明路径被找到则结束这一步并执行步骤 4。遍历完所有邻格后，如果没有找到目标格，而且这个时候开启列表已经空了则说明路径不存在则结束查找，退出。否则执行步骤 2。

4. 保存关闭列表中所存储的最短路径。

2.2 弗洛伊德路径平滑算法

弗洛伊德算法是解决任意两点间的最短路径的一种算法，能够正确解决有向图或存在负权的图的最短路径问题，同时也常常被用于计算有向图的传递闭包。弗洛伊德算法的时间复杂度为 $O(N^3)$，空间复杂度为 $O(N^2)$。

弗洛伊德算法的主要思路如下：从任意节点 A 到任意节点 B 的最短路径存在 2 种情况，要么是直接从 A 到 B，又或者是从 A 通过了若干个节点 X 之后再到 B。所以，我们假设 d(AB)为节点 A 到节点 B 之间存在的最短路径的距离，对于节点 A 与节点 B 之间的每一个节点 X，d(AX)是节点 A 到节点 X 的距离，d(XB)是节点 X 到节点 B 的距离，判断是否存在 d(AX) + d(XB) < d(AB)，如果存在，则可以证明从 A 到 X 再到 B 的路径比 A 直接到 B 的路径短，我们便设置 d(AB) = d(AX) + d(XB)，这样一来，当我们遍历完所有节点 X，d(AB)中记录的便是节点 A 到节点 B 的之间的最短路径的距离。

其算法描述：

1.对于目标节点，两点直达的距离是已知的，则首先将这两点的距离设置为该距离，如果不能直接地从一点到达另一点，那么将两点的距离设为无穷大。

2.对于包含目标节点在内的每一对顶点 u 和 v，看看是否存在一个顶点 w 使得从顶点 u 到顶点 w 再到顶点 v 的路径长度比已知的路径更短。如果存在就将这条路径设为最短路径和并将距离更新。

3.遍历完所有点后，所剩下的最短路径即为所求最短路径，最短路径长度即为该路径的距离。

弗洛伊德路径平滑算法是一种在平滑算法内部应用了弗洛伊德算法的平滑算法。

弗洛伊德路径平滑算法试用的是通过 A*寻路算法得到的最短路径后进行平滑，它的主要步骤只有两步：一、如果路径中存在相邻的点共线，那么就将共线的点进行合并，只留下端点；二、去掉路径中存在的多余拐点，如果能够直接到达，就直接到达，不经过这些拐点。这个过程如下图所示：

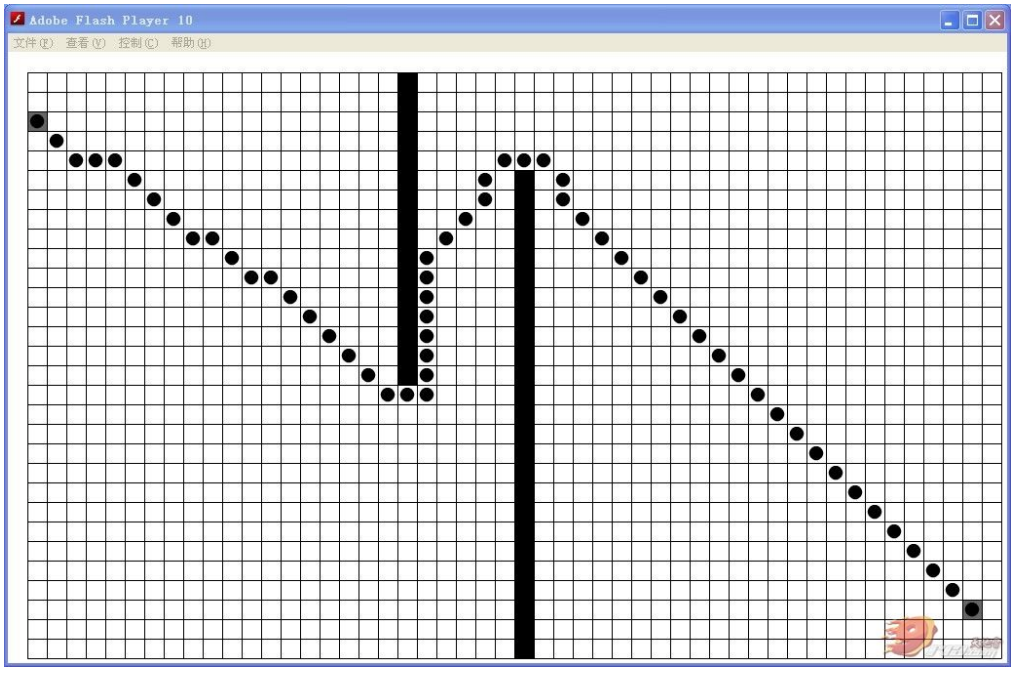


图 3

去掉共线点

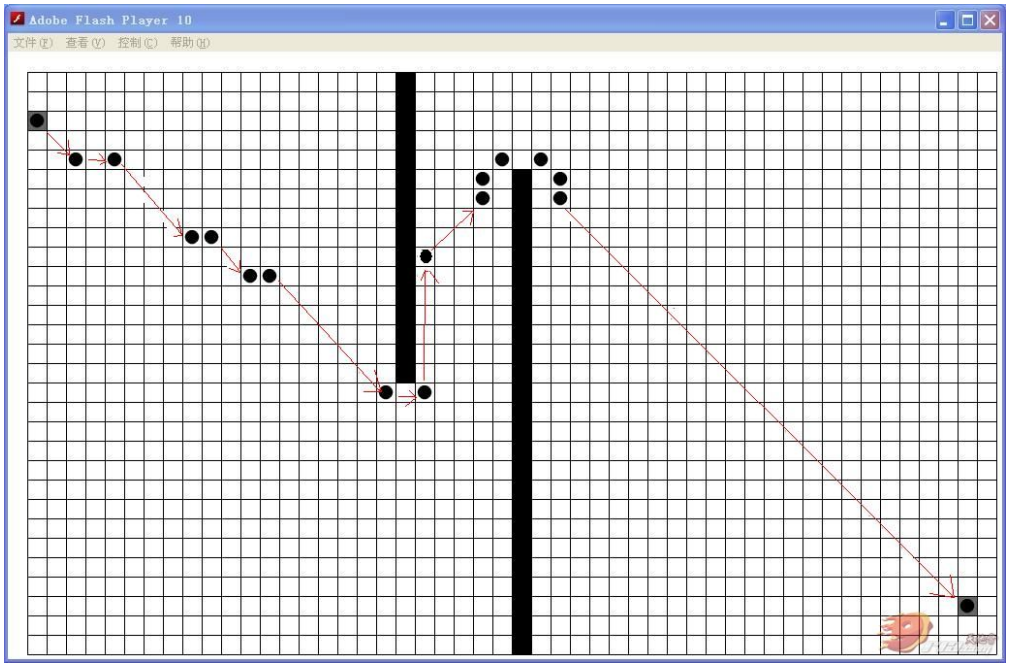


图 4

去掉多余拐点

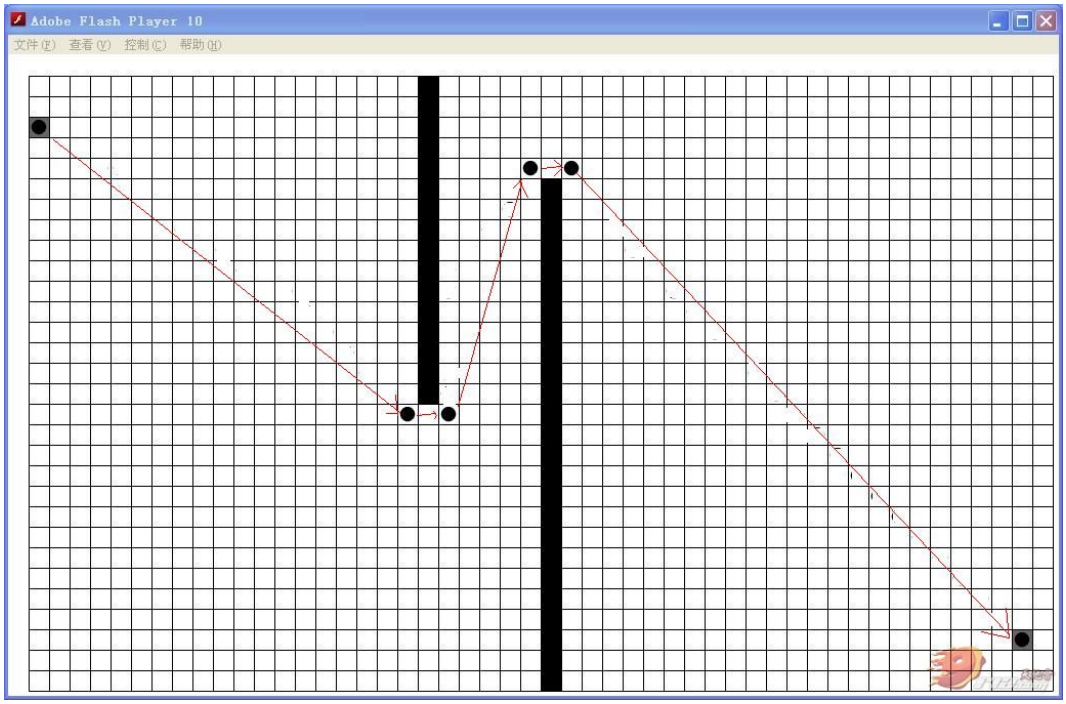


图 5

可以看到，使用弗洛伊德路径平滑算法处理后的路径表现的与我们所期望的一致，路径中存在的节点大大减少，而且路径不存在多余的拐点，能够直接到的就直接到了。

判断几点共线的主要原理在于，如果在平面直角坐标系中存在三点 A(x_1, y_1)，B(x_2, y_2)，C(x_3. y_3),这三点的横坐标和纵坐标满足以下关系：

$x_2 - x_1 = x_3 - x_2$ $y_2 - y_1 = y_3 - y_2$

则说明 A,B,C 三点共线，那么只需要保留处于端点的两点，除去中间的点。对于空间直角坐标系，原理类似，$A(x_1, y_1, z_1)$,$B(x_2, y_2, z_2)$,$C(x_3, y_3, z_3)$,当存在以下关系时，A、B、C 三点共线：

$x_2 - x_1 = x_3 - x_2$ $y_2 - y_1 = y_3 - y_2$ $z_2 - z_1 = z_3 - z_2$

这种情况下也可以消除这三点中的中间点。

对于多余拐点的消除，原来主要是：若路径中存在的节点一系列 A,B,C,D,E,F,G，且节点 A 和节点 G 直接地连线所经过的节点中没有一个节点是不可通过的节点，那我们可以称节点 A 与节点 G 之间不存在障碍物。如果两格节点之间不存在障碍物，那么这两个两点间的所有其他节点都是可以被除去的。例如上述 A,B,C,D,E,F,G 这些节点，假设节点 A 与节点 G 之间不存在障碍物，那么我们将 A 与 G 之间的 B,C,D,E,F 节点全部都除去，最终形成的路径将只剩下 A 与 G 两个节点。

2.3 DDA 算法

DDA 法即数值微分法(Digital Differential Analyzer)，是一种基于直线的微分方程来生成直线的方法。

算法描述：

设(x_1, y_1)和(x_2, y_2)分别为所求直线的起点和终点坐标，由直线的微分方程得：

$$d_x/d_y = (y_2-y_1)/(x_2-x_1) = m = \text{直线斜率} = \Delta y/\Delta x \quad (1)$$

那我们就可通过计算 x 方向上的增量Δx引起 y 的改变来生成所需的直线。

$$x_{i+1} = x_i+\Delta x \quad (2)$$

$$y_{i+1} = y_i+\Delta x * m \quad (3)$$

我们也可通过计算由 y 方向的增量Δy引起 x 的改变来生成直线，但是这比改变 x 方向的增量少见：

$$y_{i+1} = y_i +\Delta y \quad (4)$$

$$x_{i+1} = x_i + \Delta y /m \quad (5)$$

我们 x_2-x_1 与 y_2-y_1 中较大者作为步进方向即选择 m 与 1/m 中较大的一个作为直线的斜率。(假设 x_2-x_1 较大)，取该方向上的增量为一个象素单位(Δx =1)，然后利用式(1)计算另一个方向的增量(Δy =Δx*m =m)。通过递推公式(2)至(5)，把每次计算出的(x_i+1，y_i+1)经取整后送到显示器输出，则得到扫描转换后的直线。

之所以取 x2－x1 和 y2－y1 中较大者作为步进方向，是考虑沿着线段分布的象素应均匀。

在空间直角坐标系中，斜率的选取基本一致，主要是将斜率变为与直线与平面的倾斜程度。

三、抖动消除

在了解抖动发生的原因之前这之前，我们需要首先了解 3D 游戏的发展之路以及一些基本概念。

“游戏”一词泛指棋类游戏例如象棋和《大富翁》；纸牌游戏，例如梭哈和二十一点；赌场游戏例如轮盘和老虎机；军事战争游戏、计算机游戏、孩子们一起玩耍的游戏。在计算机的语境下，“游戏”一词会使我们在脑海中浮现出一个虚拟世界，玩家可以控制人物、动物或玩具。

绝大部分游戏是软实时互动基于代理计算机模拟的例子。

在电子游戏中，会用数学方法来为真实世界的子集建模，从而使这些模型在计算机中运行。显然，这些模型只能是显示或者想象世界的简化或者近似版本，因此，数学模型是现实或者虚拟世界的模拟。

基于代理模拟是指，模拟中多个独立的实体（称作代理）一起互动。

所有的互动游戏都是时间性模拟的，即游戏世界是动态的——随着游戏事件和故事的展开，游戏的状态随着时间改变。游戏也必须响应人类玩家的输入，这些输入是游戏本身不可预知的，这也说明游戏是互动时间性模拟的，大多数游戏会实时回应用户输入，这即为互动实时模拟。

软实时是指即使错过期限却不会造成灾难性的后果。所有游戏都是软实时的。模拟虚拟世界需要用到很多数学模型。数学模型分为解析式和数值式。例如，一个刚体因为地心引力而以恒定加速度落下，其分析式数学模型可写为：

y(t)= gt2 +v0t+y0

分析式模型可为其自变量设任何值来求值。例如上式，给予初始条件 v0 和 y0、常量 g，就能设任何时间 t 来求 y(t)的值。在电子游戏中，用户的输入是不能预知的，因此不能预期对整个游戏完全适用分析式建模。

刚体受地心引力落下的数值式模型可写为：

y(t+ t)=F(y(t),y`(t),y``(t),...)

也就是说，该刚体在（t+ t)未来事件的高度，可以用目前的高度、高度的第一导数，高度的第二导数及目前的时间 t 为参数的函数来表示。为实现数值式模拟，通常需要不断重复的计算，以决定每个离散时间的系统状态。游戏也是如此运作的，一个主游戏循环不断执行，在循环的每次迭代中，多个游戏系统，例如人工智能、游戏逻辑、物理模拟等，就会有机会计算或者更新其下一个离散时间的状态。这些结果最后可渲染成图形显示、发出声效或者输出至其他设备。

游戏引擎这个术语在 20 世纪 90 年代中期形成，这与第一人称射击游戏如 id software 公司的《DOOM》有关。《DOOM》将其软件构架划分为核心软件组件（如三维图形渲染系统、碰撞检测系统和音频系统等）、美术资产、游戏世界、构成玩家游戏体验的游戏规则。这样的划分非常有价值，另一个开发商取得了这样的游戏的授权之后，只需要制作新的美术、关卡布局、武器、角色、游戏规则等，对引擎软件做出很少的修改，就可以把游戏打造成新产品。

现在主流、常见的商业引擎有：Value 公司的 Source 引擎，Epic 的 Unreal 引擎，以及在移动端很常见的跨平台商业引擎 Unity3D，Cocos2dx 等。另外还有很多游戏公司有自己专用的私有引擎。

一般认为，首个三维第一人称射击游戏是《德军司令部》。这款游戏有美国的 id software 于 1992 年制作，他引领游戏产业进入到了令人们兴奋的领域。Id software 又相继开发了《DOOM》、《Quake》等游戏。随着 20 多年的发展，3D 游戏已经变得非常常见，基本上已经是随处可见。现如今的游戏大多数是 3D 游戏。

3.1 抖动与抖动发生的原因

在 3D 游戏开发的过程中，当在游戏内对游戏中的对象进行移动、转向时，游戏的画面会发生剧烈的抖动，会使玩家晕眩，影响玩家体验。

要明白抖动的发生，首先需要明白电子游戏是怎么运作的。首先，需要认识到电子游戏也仅仅是一个可编译和运行的程序。就像其他的程序一样，main 函数也是它第一个被执行的函数。游戏的结构如下图所示：

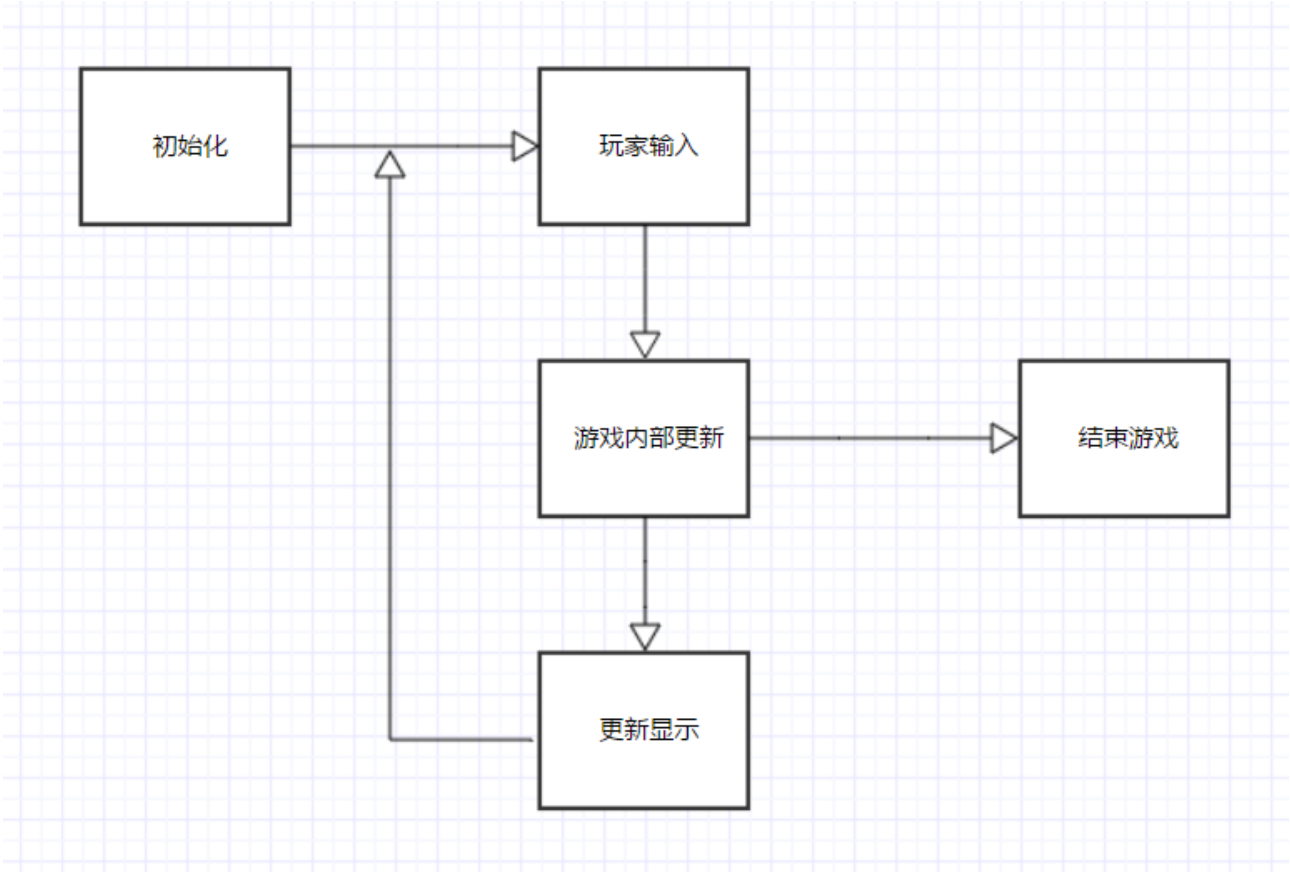


图 6

上图包含了五种状态，电子游戏最重要的三种状态在循环中互相连接，这个循环通常称作游戏循环。游戏循环式进行玩家输入，更新游戏内部数据，更新显示的地方。

初始化是尤其开始的地方，通常包含了启动动画以及一个包含了音量、画质等选项的选项菜单，在这个过程中会进行一些必要参数的设定，以及申请内存，设置堆栈，检测游戏环境以及加载显示驱动（比如包含集显和独立显卡的机器来加载独立显卡）。在这些都已经设置完毕后，玩家选择了开始游戏，则跳转到下一个状态。

玩家输入是对玩家所使用的输入设备例如键盘、鼠标、游戏手柄等进行监听，对玩家的输入的信息进行采集，并且在游戏内部中以游戏能够处理的形式进行存储。供后一个状态使用。

更新游戏内部是游戏的心脏，游戏根据玩家的输入，来改变游戏中的设定值，例如对玩家操作的人物进行移动，玩家游戏中的敌人的运动与反应，同时准备好显示所需要的所有数据与图像。

更新显示是将计算后的画面投射到屏幕上显示出来。一般来说，你可以选择对每个物体挨个在屏幕上绘制或者说按照例行办法，先把所有的对象计算好，然后进行一次性的绘制。在屏幕上进行绘制是一个花费时间比较多的过程。所以一次性进行绘制将会有比较好的性能。

结束游戏是玩家选择退出游戏或者玩家通关之后的一种正常状态。一般会有一个结束动画或者提示语来告诉玩家游戏已经结束了。同时也会进行数据的保存，释放内存，消除堆栈等工作。

摄像机指的不是现实社会中用来照相的那个摄像机，但是两者发挥的作用基本上是一样的。电子游戏中的摄像机用另外的一个词来说是“视角”，也就是看游戏世界的角度。展现在玩家屏幕上的画面便是通过摄像机的“照”出来的画面。

由于玩家的画面是由摄像机照出来的，那么画面抖动会有两个原因：游戏世界在抖动或者是摄像机在抖动。 游戏世界我们没有进行直接操作，那么按道理来说不会发生抖动，这个可能性就可以被排除掉。那么抖动发生的原因在于摄像机在抖动。

开发游戏时，当对游戏内部的对象进行移动、旋转操作时时，由于游戏本身的设定，摄像机也会跟随着玩家移动的对象进行移动、旋转，这在酷跑类游戏中很常见。由于游戏是帧驱动的，游戏循环每执行一次，游戏就播放一帧。每一帧中都会执行摄像机位置的变换，使之看向所需要看向地方，摄像机不停变换位置，而且是无规则的。不规则的位置变化会引起抖动。

3.2 抖动消除原理

既然画面的抖动是由于摄像机的抖动造成的，那么，我们要消除画面抖动，就只需要消除摄像机的抖动就可以了。画面抖动都发生在画面移动时即摄像机朝向或者位置发生变化时。由于现今的游戏通常都是帧驱动的游戏，我们需要在游戏循环每次执行的时候，让摄像机位置、朝向的改变变得平滑也就是使得摄像机看向的位置移动变得平滑，这样，在游戏的过程中，我们所看到的画面是在进行平滑的变换，就如同我们自身走动一般。

抖动消除有一个前提，时间上要是平滑的。也就是说要么是每一帧执行所用的时间是一样那要么就处理成一样的效果。现代游戏是帧驱动的，我们不能保证每一帧所执行的时间是一致的，但是我们可以获得每一帧执行所消耗的时间，在处理时，我们以消耗的时间为影响因子乘如，这样便实现了时间上的平滑。

再者是路径上的平滑，平滑无非两种，一种是延迟，一种是根据以前的速度进行加速度计算，然速度变化变慢。

本文中消除抖动的平滑算法分为两部分：直行时平滑部分和转动时平滑部分。分别对应在直行时位移所导致的抖动和转向时旋转导致的抖动。

3.3 算法实现

3.3.1 不转向时平滑算法

在直行时，基本上不会涉及到摄像机朝向的变化，也基本上不会导致摄像机旋转。那么我们可以使摄像机朝向某一个确定的方向，控制摄像机沿着平滑的路径，以平滑的速度运动。

对于平滑的路径，我们使用 A*寻路找到这样的路径。采用以下步骤：

1. 开启列表置为空，将起始格添加到开启列表之中。

2. 寻找开启列表中当前 F 值最低的格子，即为当前格。然后将当前格加入到关闭列表。

3. 对于当前格相邻的每一个格子，如果这一格不可通过或者已经在关闭列表中，那么就略过这一格，比较下一个格子从 v；如果这一格不在开启列表中，那么就把这一格添加到开启列表中去，并且把当前格作为这一格的父节点，并记录这一格的 F,G,和 H 的值；如果这一格已经在开启列表中，则检查这一格的 G 值是否更低，如果是，就把这一格的父节点改成当前格，并且重新计算这一格的 G 和 F 值；如果你把目标格添加进了关闭列表，则说明路径被找到则结束这一步并执行步骤 4。遍历完所有邻格后，如果没有找到目标格，而且这个时候开启列表已经空了则说明路径不存在则结束查找，退出。否则执行步骤 2。

4. 保存关闭列表中所存储的最短路径。

该过程的伪代码为：

```
开启列表 ← 起始格
关闭列表 ← {}
repeat
    当前格 ← min(F(开启列表))
    foreach 相邻格(当前格) do
        if 相邻格(当前格) in 关闭列表 or 相邻格(当前格)  no access then
            skip
        end
        if 相邻格(当前格) not in 开启列表 then
            开启列表 ← 相邻格(当前格)
```

```
        end
        if 相邻格（当前格） in 开启列表 then
            if G(P(相邻格(当前格)))< G(当前格) then
                当前格 ← 父节点(相邻格(当前格))
                renew(F(相邻格(当前格)))
                renew(G(相邻格(当前格)))
            end
        end
    end
until 目标格 in 关闭列表 or 开启列表 = {}
save path
```

使用 A*寻路算法之后，我们得到了一条摄像机移动的路径。这个路径是由一系列的点组成的。对于这些点，可能有重复的点,我们需要使用弗洛伊德路径平滑算法来对结果进行简化、平滑。

算法过程：

对于初识连续的点 A_1， A_2， A_3… A_n 中的某个点 A_i (x_i， y_i， z_i),如果前一点 A_{i-1} (x_{i-1},y_{i-1},z_{i-1})和后一点 A_{i+1} (x_{i+1}， y_{i+1}， z_{i+1}),如果 $x_i-x_{i-1}=z_{i+1}-z_i$ 且 $y_i-y_{i-1}=y_{i+1}-y_i$ 且 $z_i-z_{i-1}=z_{i+1}-z_i$ 那么这三点共线，则在点集中除去 A_i 点。将所有共线点处理后得到新的点集 A_1 ,A_2， A_3,.. A_x,如果 A_j， A_{j+1}， … A_{j+m} 中的点都不是不可移动点且 A_j 到 A_{j+m} 之间没有障碍物，则删除 A_{j+1} 到 A_{j+m-1} 的所有点。

伪代码如下：

```
初识点集 ← {A1,A2,A3….An}
i ← 1
repeat
      if Ai,Ai-1,Ai+1 共线 then
            delete Ai
      end
      i ← i+1
until i = n
i ← 1
repeat
      if Ai,Ai-1,Ai+1 都可以移动 and Ai， Ai+1 间没有障碍 then
```

```
                delete Ai
            end
        until
```

通过上述方法，得到了平滑的路径。

得到了平滑的路径之后，我们需要对速度进行处理。由于游戏是一帧一帧进行执行，每一帧所耗费的时间可能不一样，因此，摄像机移动的距离不能简单的设置为一段距离，应当以一段距离乘以一帧执行的时间。距离选取不应太长，否则会出现移动不连贯，出现跳帧的现象。

如果在运动的过程中有速度变化，则需要有一个连续的速度变化过程，不能瞬间就变化到某个速度去。例如启动运动的时候，游戏中物体会由静变化到动，这在这个过程中，需要有一个加速的过程。可以采用一个恒定的加速度。

3.3.2 转向时的平滑算法

2D 游戏中由于只有 x,y 两根坐标轴，只会在平面上进行移动，不会存在转向的问题。但是在 3D 游戏之中，很有可能会有转向。在这种情况下，摄像机的朝向会发生改变，朝向变化也需要进行平滑。

转向时平滑需要考虑的是自然的转向不是就在原地为圆心做圆周运动，而是以原地以外的某个点为圆心做圆周运动，因此，在转向时，需要进行处理。通过圆的二维平面方程：

$$(x-a)^2+(y-b)^2= r^2$$

得到圆的轨迹。摄像机的朝向与摄像机坐标与圆心的方向相同。

此时运动的长度可以根据角度来。相同时间转动的角度应该是确定的。则根据转过的角度 θ 来得到每一帧处于的点和朝向。设某点为(x, y, z),为了简化处理，假设圆心为(0,0,0),半径为 r，在平面 Ax + By = 0 上画圆，单位时间转动的角度值为 α，则新一点的坐标值为

$$x_1 = x \pm r[1-\cos(\alpha\Delta t)]$$

$$y_1 = y \pm r(1-\sin\alpha\Delta t)$$

$$z_1 = z$$

其中+号还是-号取决于方向。

对于一般情况下转向，就只简单的做一下数学上的推导，在实际游戏开发中，一般都会选择 Ax+By=0 这个平面为基准面。

在三维空间中，设圆的圆心为(x_0, y_0, z_0),圆的半径为 r,圆的法向量为(α, β, γ)，则以(x_0, y_0, z_0)为圆心以 r 为半径的球面方程为

$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2=r^2$$

圆所在的平面方程为：

$$(x-x_0,y-y_0,z-z_0)\cdot(\alpha,\beta,\gamma)=0$$

联立两式得到空间中圆的方程：

$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2-\alpha(x-x_0)-\beta(y-y_0)-\gamma(y-y_0)=r^2$$

设之前的点为(x, y, z)，单位时间内转动的角度值为 μ，则有方程：

$$(x-x_0,y-y_0,z-z_0)\cdot(x_1-x_0,y_1-y_0,z_1-z_0)=\mathrm{r}^2\cos(\mu\Delta t)$$

$$(x_1-x_0,y_1-y_0,z_1-z_0)\cdot(\alpha,\beta,\gamma)=0$$

$$(x_1-x_0)^2+(y_1-y_0)^2+(z_1-z_0)^2=r^2$$

联立上式，即可得到新的点的坐标。

四、实验与分析

4.1 实验

我们在所开发的 demo 为一个酷跑类型的游戏，游戏过程中，玩家控制游戏中的角色一直前进，中途可能会转向会有障碍物。摄像机在游戏过程中会一直跟随着玩家来一起运动。游戏 demo 使用的是 Unity3D 引擎来制作。

实验一：游戏中摄像机做匀速直线运动。

实验结果：游戏正常运作，不发生抖动。

实验二：摄像机速度发生变化例如从静到动。

实验结果：在应用平滑算法之前，速度突然变化，画面剧烈抖动。应用平滑算法后，游戏速度有一个加速过程，画面移动没有发生抖动。

实验三：摄像机横向平移。

实验结果：应用平滑算法前，摄像机横向移动时，摄像机前进方向速度突然变化，横向速度突然变化，画面偶尔会失控，剧烈抖动。应用平滑算法之后，前行速度以恒定加速度减慢，横向速度以恒定加速度加快，平缓移动，未发生画面抖动。

实验四：摄像机转向。

实验结果：应用平滑算法之前，摄像机会突然停止前行并且以摄像机当前所在的位置为圆心，将朝向进行旋转。应用平滑算法后，摄像机不会停止，而是摄像机会以转向处的外的某个点为圆心，以匀速做圆周运动转过，同时摄像机的朝向也逐步变化。屏幕上会展示出沿途的物体，转到了合适的方向之后，摄像机又开始执行，继续前进，等待玩家的指令输入。在这个过程之中，摄像机移动平缓，画面稳定，没有发生抖动的情况。

4.2 分析与讨论

在应用算法前，匀速之前运动的时候，由于不存在速度上和路径上的不平滑，所以游戏过程中表现非常正常，没有出险抖动、跳帧、失控等不正常现象。但在加速、减数时，由于此时速度发生了变化，由于游戏没有对速度进行处理、平滑，所以速度的突然

变化会导致游戏画面的抖动、跳跃。当有横向移动的时候，之前会直接瞬间横向移动，缺少了速度变化的过程，路径也不平滑，摄像机照出来的场景变化大、频繁，导致了抖动和跳帧。当转动时，最开始的是瞬间改变朝向，这样会发生非常剧烈的抖动，还可能会是摄像机失控。后面改为以原地为圆心旋转。但是这使得摄像机照出的画面显得很生硬，不自然。与现实差距较大。

在应用平滑算法之后，匀速直线运动由于路径、速度都是平滑的，结果与之前一样是在预料之中的，也是符合情景的。在加速、减速的直线运动时，之前由于速度不是平滑的，所以对速度进行平滑处理后，摄像机运动的速度有了一个平滑的变化，游戏画面变化不突兀、没有发生抖动也是符合预期结果的。转向时改变后游戏变化连续，未出现抖动现象，符合预期结果。横向移动时，可以感觉到前后速度与横向速度的变化，未出现抖动，符合预期结果。

五、总结与展望

通过应用平滑算法，使得游戏中摄像机能够平滑、稳定的进行移动、旋转，使得游戏的画面能够连续、稳定的进行变换，不会出现游戏画面的抖动。

本文所使用的平滑算法，在匀速直线运动时不会导致异常，在转向、变速运动、横向移动时，能够很好的解决画面抖动与画面不连续的问题，说明了本文的算法是较为可靠的。使能够应用到开发过程中的。

参考文献

[1] 穆俊. 计算机游戏设计原理以及游戏引擎的设计思想[J]. 硅谷,2014,(03):49+98.
[2] 袁俊杰,徐小良. 基于漏斗的实时 VBR 视频最短路径平滑算法[J]. 计算机系统应用,2010,(07):42-46.

声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得四川大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

本学位论文成果是本人在四川大学读书期间在导师指导下取得的，论文成果归四川大学所有，特此声明。

学位论文作者（签名）_____

论文指导老师（签名）_____

二〇一七 年 月 日

致谢

在本文的编写过程中，感谢我的指导老师陈杰老师的细心指导，老师在从设计拟题、文章的编写、文献的选取都提供了很大的帮助，在过程中每周五都会在望江实验室进行交流、了解进度、对不清楚的部分进行指导、说明。这篇文章的成形，离不开老师的心血。在此，谨向导师表示崇高的敬意和衷心的感谢!

通过这篇论文的撰写，使我能够有机会和精力来系统、全面的学习有关游戏开发的理论知识，这对于我今后的工作和和我以后自身的发展，是一笔极其宝贵财富。由于本人理论水平比较有限，论文中的有些观点和归纳和阐述难免有疏漏和不足的地方，欢迎老师和专家们指正。

附录

附录一 文献翻译

摘要

在现今的社会中，电子游戏已经变得随处可见。在这边文章中，我将探讨是什么使得 3D 游戏能够出现。要想理解 3D 游戏，首先了解三维图形的基本知识。在开发一个电子游戏之前，你首先需要知道电子游戏是怎么运行的。游戏开发过程中，需要遵循分层开发的原则。电子游戏的分层结构表明了它是怎么构建，这是电子游戏开发中不可避免的一部分。

一、前言

当今的世界上，电子游戏变得随处可见。同其他的计算机软件一样，电子游戏也已经发展了好多年了。对于电子游戏来说，最大和最值得注意的挑战是使显示的画面显得很生动，富有智慧和生气。最开始的电子游戏都是 2D 的，例如在二维平面上飞行的飞机。

现代的电子游戏几乎都是 3D 游戏。计算机产业的技术进步使得从 2D 到 3D 的这一步跨越成为了可能。在这篇文章中，笔者将探讨是什么使得 3D 游戏成为了现实。在这之以前，我有如下问题：

- 开发一个 3D 游戏需要什么？
  - 3D 图形的基本知识有哪些？
  - 电子游戏的基本知识有哪些？
  - 如何将前面说的应用到 3D 游戏之中？

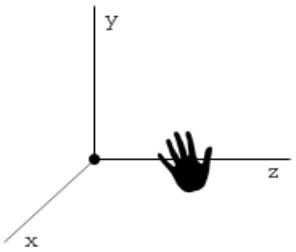
笔者将会为每一个子问题开一章单独进行详述。在总结一章中，我将会总结所有的材料然后为上述问题给出答案。

二、三维图像的基本知识

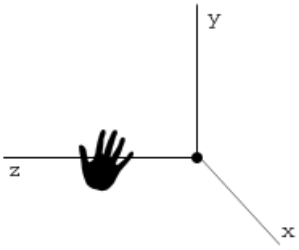
根据 Watt et al, 2001，计算机三维图像在电子游戏中扮演着支持那些用于构建游戏世界、角色和对象，并且使它们能够互相作用并且能够将其投影到二维屏幕上的工具的角色。只有对三维空间有着数学基础才能实现上面的功能。在这一章中，笔者将会简单阐述相关数学基本知识，但是不会做详细的数学理论探讨。这仅仅是为帮助理解 3D 游戏中的 3D 图形的知识。

2.1 三维坐标系

首先从三维坐标系开始。该坐标系有三个左边：x,y,z。通常，有两种方式来表示这种坐标系：左手坐标系和右手坐标系（图一与图二）。



图一 左手坐标系



图二 右手坐标系

如果想要检测你正在是用什么坐标系，只需要用手握住 z 轴，如果大拇指与 z 轴方向相同，那么你用的那只手，就是哪种坐标系。

虽然数学上惯例使用右手坐标系，但是实际应用者，二者都有使用。二者唯一的差别就是 z 轴的指向的方向相反。

2.2 对象

坐标系已经有所了解，那就该了解下一项：对象。在最底层的实现中，三维对象被表示为坐标系中的一系列点。对于简单的对象，你可以在三维空间中定义一些点，通过在点之间连线来构建对象。但是对于复杂的对象来说，这是不可行的。因此，你需要一种不同的方法来表现三维的对象。(Watt et al, 2001)中提到了以下主流计算机图像模型。

1.多边形模型

对象一系列的多边形来填充形成立体。在 (Angel, 2003)中，多边形被定义为使用一系列的线形成的具有边界的闭合对象，但是其拥有内部空间。这也是其有时也指内部填充区域的原因。通过多边形来描述物体，可以使得物体表现时有着不同的精度。使用的多边形越多越复杂，物体表现也就越准确。

2.双三维参数块

首先应该了解一下“曲面四边形”的概念。一般来说，这个概念类似于多边形填充只不过在这里，单个多边形是一个曲面。每一小块都是通过数学公式来计算出其在三维空间中位置和形状的。公式使得我们可以计算出块表面上的所有的点。块的形状可以通过修改数学公式来改变。这导致了强大的相互作用，意义重大。当某一个块的形状发生改变时，保持该块与邻接的块的平滑是一个重要的问题。双三维参数块可以是一个准确或者是近似的表现。对于自身，他们只能准确的表现。这意味着，一个物体的形状如果与块的形状精确的一致，那么，该物体只能被准确的表达。这种令人纠结的状态是必须的，因为表现真实对象时，物体实际形状未必与表面一致。

这种模型的最大优势在于它的花费。与多边形模型相比，创意同一个对象，双三维参数块花费使用的元素数量更少。

3.立体几何

这是对模型对象在刚性限度内的准确表示法。它有意识到很多高级对象可以表示为初级形状或几何基元的组合。例如，一个有一个洞的金属块可以表示为一个长方体和圆柱体的差集。将其联系起来，我们可以看到这样表达有利于进行简单和直观的控制——如果一个金属板上有洞，那么就可以定义一个圆柱体，将其与一个长方体做减法来得到这样的一块板。

立体结合法是一个体积表现法：形状是由初级形状或几何基元的组合而来。这与上面两个方法使用表面不同。

4.空间剖分技术

通过这项技术，对象空间被分割为基本的立方体——体元，每一个体元要么被标记为空要么被标记为对象的一部分。这类似于二维空间中的像素。如果将所有的三维对象空间都进行标记，那么花费是非常大的。但是它在计算机图形学中得到了很好的应用。

这个模型使用三维对象占用的空间表现，最前两种方法使用对象的表面来表现。

5.隐式表达

隐函数偶尔在文章中作为一种对象表示形式被提到。

隐函数是球体的定义法，例如：

$x^2+y^2=z^2$

他们本身的作用很有限，因为很少有对象能够按照这种方法来表现。就渲染的形式而言，这种方式也不方便。但是，应当注意到，这种表现形式经常在三维计算机图形学中,尤其是在球体使用频繁的光线追踪,对象本身和边界都是多边形网格表示。

隐式表达可以延伸到隐式函数，可以松散地描述为对象由数学上定义的一个受底层参数影响的表面的集合例如球。隐式函数在变形动画中发挥着作用。对于表现真实的物体，它的作用很小。这种方法展现了它在塑造有机形状上的潜力。元球是一个隐式的造型技术,用于创建上述有机形状。

在所有的对象模型中，多边形模型是实际的标准模型。主要原因在于该模型对 CPU 要求比较低，然而会损失一部分的精度。但是其他的模型也开始在 3D 游戏中变得越来越常见。虽然这些模型对 CPU 要求更高，与多边形标准相比，他们的优点已经足够去使用它们了。CPU 的发展和内存的进步也促进了这些模型的发展。

2.3 变换

在(Watt et al, 2001)中，变换被形容为三维场景产生的重要工具。其被用作将对象在环境中四处移动以及创造用于显示的二维平面。变换背后的基本原理是将一个点以某种方法映射到另一个点。不同的变换对应的方法不同。计算机图形学中用到的变换称为仿射变换。根据(Weisstein, 2004)，任何保留共线性（开始前所有点在一条之前上的，变换后依然在一条直线上）和长度比例（某个点变换

前是线段的中点，变换后依然是中点）的变换是仿射变换。在这种情况下，仿射是一种不会从无穷远移动的特殊的投影变换。从计算机图形学的角度来说，变换只会在预定义的三维坐标系中进行。

仿射变换可以使用矩阵来表示，许多的仿射变换的结合可以使用矩阵运算来表示。这使得对一个对象进行多种变换成为可能。想要了解更多仿射变换地数学背景和矩阵，可以参见(Angel, 2003)。

在计算机图形学中，仿射变换最常见和最基本的用途是平移、旋转和缩放。因为剪切的重要性，在(Angel, 2003)中提出了剪切也是一个基本的仿射转换。但是由于剪切可以通过上述三种方式组合来实现，所以我不认为它是一种基本的仿射变换。我会给出平移、旋转和缩放的简单定义，这些定义源自(Angel, 2003)。

平移是将所有点点沿某个确定的方向移动相同的距离的操作。由于我们是在三维坐标系中进行处理，所以平移将会有三个坐标方向。

旋转是以一个固定点为圆心，沿着某个向量移动特定角度。 对于一个给定的不动点有三个自由度数：两个角指定的向量方向和一个角指定旋转的角度。

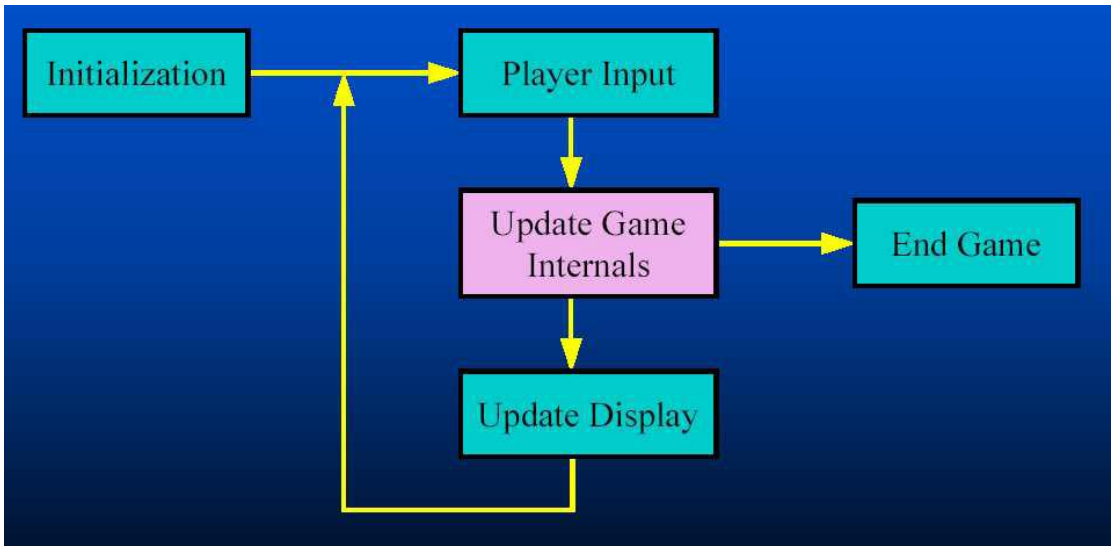
缩放用于在三维空间中放大或者缩小对象。缩放变换有一个不动点。你需要确定这个不动点和缩放方向以及缩放因子来进行缩放。

2.4 总结

要理解 3D 游戏，三维图形的知识是必须的起点。首先，需要定义一个三维坐标系，再者，你需要确定怎么在坐标系中来表现你所选取对象。你可以选取不同的对象模型，当然有取也有所得。目前，多边形模型是事实的标准，但这正在发生着改变因为，cpu 变得越来越快，内存也变得越来越多。最后，你需要有一种方法在坐标系中操作你的对象。这可以通过仿射变换来实现。最常用和最基本的三种仿射变换是平移、旋转和缩放。其他的仿射变换都可以通过这三种按照特定的顺序组合出来。

三、初识电子游戏

为了制作一个电子游戏，你需要明白电子游戏是怎么运作的。首先，你需要认识到电子游戏也仅仅是一个可编译和运行的程序。就像其他的程序一样，main 函数也是它第一个被执行的函数。Main 函数的结构如图三所示的流程图一样。



图三

上图包含了五种状态，电子游戏最重要的三种状态在循环中互相连接，这个循环通常称作游戏循环。在(Howland, 2001)中，游戏循环被描述为获取输入，输出给玩家，更新游戏的一系列程序。让我们看看每个状态和它的意义：

• 初始化（开始游戏）

按照(Howland, 2001)的说法，初始化是：“每个游戏的开始部分，通常包含了一个动画序列来接受游戏的故事或者背景和选项目录来开始游戏或者改变游戏的一些参数。这些参数通常包括音量、图像选项，多玩家选项、难度和开始的等级等”。

我同意，当你玩游戏的时候，游戏是已经开始了。但是如果深思，这些事情在游戏循环中都已经完成了，因为玩家输入能够成功，游戏也能以一种合适的方式做出响应。这些改变需要在循环内进行更新并且输出给玩家。我认为，初始化将必要的参数都设置了，必要的函数都调用了等等于是，游戏可以由初始化转换到游戏循环中去。初始化一般会做检查、请求内存，设置栈，加载显示驱动等。

• 玩家输入

(Howland, 2001)中有以下说法：“输入采集协程会从玩家所使用的输入设备中获得输入信息，并在游戏内以游戏能够处理的方式存储用来改变游戏内部状态。”注意，采集信息是用于下一个状态（更新游戏内部）。

• 更新游戏内部

(Howland, 2001)中提到：“游戏更新协程是游戏真正的心脏，从根据玩家的输入移动玩家的角色到敌人的动作反应已经判定游戏的输赢在内的所有事在这里决定。同时，也为显示做准备。

• 屏幕显示

根据(Howland, 2001)，有两种方式可以在屏幕上进行显示：“你可以一次性在屏幕上绘制出所有东西，或者按照通常地做法，设定好所有东西然后再进行绘制。绘制花费的时间比游戏中其他大多

数过程都要长，所以最好一次性进行操作。决定绘制那些东西将花费相对比较长的时间来进行你游戏中的所有检查。所以最好在真正的执行绘制之前将这个过程做完。

• 结束游戏

(Howland, 2001)中描述说，正常情况下，当游戏结束时，结束动画将会播放显示，或者说至少有个提示结束的。这个描述就如同游戏的开始状态的描述一样：从玩家的角度来描述了这个状态。另外，我认为其他的一些事情例如调用函数保持数值等等也是很有意义的，这使得游戏能够正常的终止。

我对游戏是怎么运作的给了一个概述。现在我们应该聚焦到现代游戏的游戏循环中所必须的一些有趣的方面。我会着眼于游戏内部的更新以及更偏重于电子游戏的玩法和人工智能。在讨论这些之前，我们先讨论一下游戏方案。

3.1 游戏方案

根据韦氏词典，方案是情节的初步草图或者主要事件。在电子游戏的内容中，方案有着一致的意义，但是需要考虑更多的东西：游戏是交互性的。

当制作游戏的方案时，有很多连接电子游戏交互元素的情节是很平常的。甚至在没有故事情节的时候，方案应当描述了游戏中可能发生的事情。对于游戏方案有以下的观点：

• 游戏方案至少描述了游戏的交互。这应当以不包含技术实现的方式尽可能详细的写下来，因为此时还没有开始编程。这些交互元素决定了游戏玩法

• 方案中应该有连接各个交互元素的过渡。方案的这部分就如同电视剧或者电影的剧本一样。现如今，绝大多数的游戏都有故事情节。一些流派的游戏可能会有相当荒诞的情节，例如实时解密游戏。但是其他的流派的游戏都是忠于剧情的，不能离开情节而存在。在这种情况下，情节与交互一样有时候甚至更重要。一个典型的例子就是角色扮演类游戏。这种情况下你需要一个专门的系统来为玩家展现故事。在(Simpson, 2002)中提到了很多种方式。例如可以使用场景或脚本。

随着游戏方案（部分）的完成，此时可以开始设计游戏的玩法。

你可以通过尽可能详细的写出来，或者（在准备工作已经很详尽的情况下）开始尝试编程做出 demo 来测试。

3.2 游戏玩法

游戏玩法是电子游戏的关键词。就如同这个词的字面意思所说，它决定了游戏怎么玩。游戏玩法被以下的因素所影响：

• 方案与游戏类型

在游戏方案中，至少交互元素要写上，因为它决定了游戏的基本玩法。在前面没有提到的是，要合理的选择游戏的类型。不同类型的游戏所包含的元素可能适合这个方案，也可能不适合。结合不同类型的不同元素来适应方案是可行的。所以，方案和类型的结合决定了游戏的基本玩法。

• 操作方式与游戏反馈

对游戏的改进、优化来说，游戏中怎么操作以及对玩家的输入很好的反馈是很重要的。

游戏的操作应该根据玩家所想来定义使之变得自然、协调。同时，应当允许高级玩家自定义输入。一个精心思考过的控制方案对游戏的可玩性很有利，因为它使得游戏更容易上手。

游戏的控制会带来反馈的问题。玩家应该可以看到他们在游戏中的行动的顺序。游戏的反应时间应当越短越好。例如，你总不希望在一个竞赛游戏中，禁赛的反应时间需要五秒钟，因为如果真的需要那么长的时间，很多事情早就发生了。

• 视角（摄像机）

尽管视角不会直接的影响到游戏的玩法，但是他的不同选择决定了游戏怎么来玩。如果玩家在游戏世界中的视角不对，玩家将很难做决定。对于 3D 游戏来说，是叫通常指的是摄像机。采用这个名字的原因，摄像机引导玩家观察游戏世界。

• 学习曲线和玩法难度

影响可玩性的最后两个因素是学习曲线和玩法难度。我将这两者放在一起的原因是两者都是关于游戏难度的等级。

学习曲线是游戏新手初入游戏时所遇到的难度。由于玩家只有通过了解了游戏的基本知识，他才能很好的玩这个游戏。因此，如果玩家对游戏了解不够，游戏的可玩性就会减少，因为缺失了必要的元素

玩法难度是你经过学习曲线之后的游戏难度。玩家已经了解游戏的基本运作元素。但这不意味着玩家已经是游戏的高手了。经过很多句的游戏之后才能达到这个程度。这通常影响游戏世界对玩家的影响。在下一节关于人工智能的章节中，我会回顾这个问题。

由于游戏的可玩性是每个游戏的核心，它决定了玩家是否会继续玩下去。就像图像是游戏的技术基石，在游戏一直存在一样，可玩性一直存在。

3.3 人工智能

根据(Kelly, 2003)，人工智能是模拟高级智慧来完成复杂工作的软件的术语。在游戏中，人工智能被开发者用于为人类的对手添加一下真实元素，提高游戏的质量。这会为玩家提供更好、更真实的游戏体验。人工智能通常用于游戏世界中玩家不能直接控制或者影响的元素。非玩家角色（NPC）是这样的一个典型例子。有很多不同的种类的人工智能技术可以用于电子游戏之中。最常用的三种是搜索算法、神经网络和有限状态机。在这一节中，我会阐述他们在游戏之中的应用。

3.3.1 搜索算法

搜索算法可以用来解决寻路问题。一个典型的寻路问题是：玩家的敌人来定位玩家的位置。

(Kelly, 2003)中讨论了搜索算法有一个问题空间模型，即搜索执行的环境。其中包含问题的一系列状态已经一系列改变状态的操作。尽管蛮力搜索算法例如广度优先算法和深度优先算法总能实现他们的目标，但是通常他们并不适合于电子游戏，因为这些算法需要消耗大量的内存以及不够高效因为要遍历大量的状态来找到目标状态。A*算法也是一个寻路算法，但是比蛮力搜索的算法要高效许多。要了解更多关于 A*算法的信息，参见(Patel, 2003)。

另一个搜索算法是极大极小值算法。他通过前一步来预测哪条路径更好。得到的信息用于选择能够给予计算机优势的移动路径。

采用哪一种算法取决于游戏。需要在速度和精确中做出取舍。对于大多数游戏，你并不是真的需要两点之间的最佳路径。你需要的只是近似最佳。你需要根据游戏中正在进行什么以及电脑的运行速度有多快来决定采用哪种算法。(Patel, 2003)谈到了 A*算法中的取舍，但是可以使得搜索算法能够一般化。

3.3.2 神经网络

当谈到神经网络的时候，第一个问题是：什么是神经网络？(Sarle et. al., 1997)给了以下答案：“神经网络没有一个得到普遍接受的定义。但是这个领域中的大部分人应该同意神经网络是有很多的简单处理单元组成的网络，每一个可能只有很少的内存。单元间通过信道连接，通常携带数字数据（而不是字符），可以被解码为各种意思。每个单元只能处理自己的数据以及通过信道收到的数据。本地操作的限制通常在训练的时候被放开”。

这个定义阐述了神经网络对游戏来说很有趣的一方面：学习的能力。(Kelly, 2003)只是标注了一下这个特征对于游戏来说是一个额外的优点就是逐步调整自己来为玩家提供更多的挑战。

对于训练规则，(Sarle et. al., 1997)有以下说法：“许多神经网络有许多种训练规则，这些规则在连接和基础数据的权重上不同。换一句话说，神经网络在例子中学习，就如同小孩子能够分辨猫和狗是因为见过了很多的猫和狗。如果用心训练，神经网络能够展现出超出训练数据的归纳能力，这意味着它们对于没有训练过的场景也可以得出一个近似的结论。

(LaMothe, 1999)给予了一下神经网络在电子游戏中的应用的例子：

- 环境扫描与分类

神经网络可使用视频、音频之类的信息来训练。这类信息可以用来选择输出反应或者来教会它输出。这种反应可以实时学习并且反应到游戏的反应中。

- 记忆

神经网络可以被游戏内部的生物用作记忆。神经网络可以通过经历一系列的回忆来学习，当然新的情况发生，神经网络可以猜测最好的应对方式。

- 行为控制

神经网络的输出可以用来控制游戏角色的行为。输入可以使游戏引擎提供的一系列参数。神经网络就可以控制游戏的角色的行为。

• 反应映射

神经网络很擅长于“联系”即从一个解空间映射到另一个解空间。联系有两种不同的层次：自我联系即输入到自身的映射和异我联系即输入与其他的联系。反应映射在后端使用神经网络创建或输出一层间接控制或行为的一个对象。一般来说，我们可能会有很多的控制变量，但是我们只会对我们教给神经网络的特定的组合有很轻微的反应。然而，在输出时使用神经网络，我们可以获取在大致相同的领域的反应。

(LaMothe, 1999)评论说上述例子看起来似乎有些模糊，事实也是这样。关键在于神经网络是一种我们可以以任意方式使用的工具。关键是应用神经网络来简化我们的 AI 编程的让游戏中的角色表现得更加智能。

3.3.3 有限状态机

在(Brownlee, 2002)中，有限状态机是这么定义的：“有限状态机（FSM）也被称作有限状态自动机（FSA），简单来说，是一个有着有限的的条件或者模式的系统或者复杂对象行为的模型，条件的变化引发模式的变化有限状态机包含四个主要元素：

• 状态。状态定义了行为和可能的结果。

• 状态转换，即从一个状态转换到另一个状态。

• 规则与条件。需要满足了才能进行状态转换。

• 输入事件。外部或者内部生成，可能会触发规则，导致状态转换。

一个有限状态机必须要有一个初始状态，这个初始状态提供了一个起点和当前状态。当前状态主要用来记忆上一个状态转换的产物。受到输入会触发触发器，会使条件来检测是否满足从一个状态转换到另一个状态。想象 FSM 的最好方式是把它看作一个流图表或一个有向图的状态,尽管可以使用技术得到更准确的抽象建模。”

图 4 展示了(Brownlee, 2002)中 FSM 的示意图。

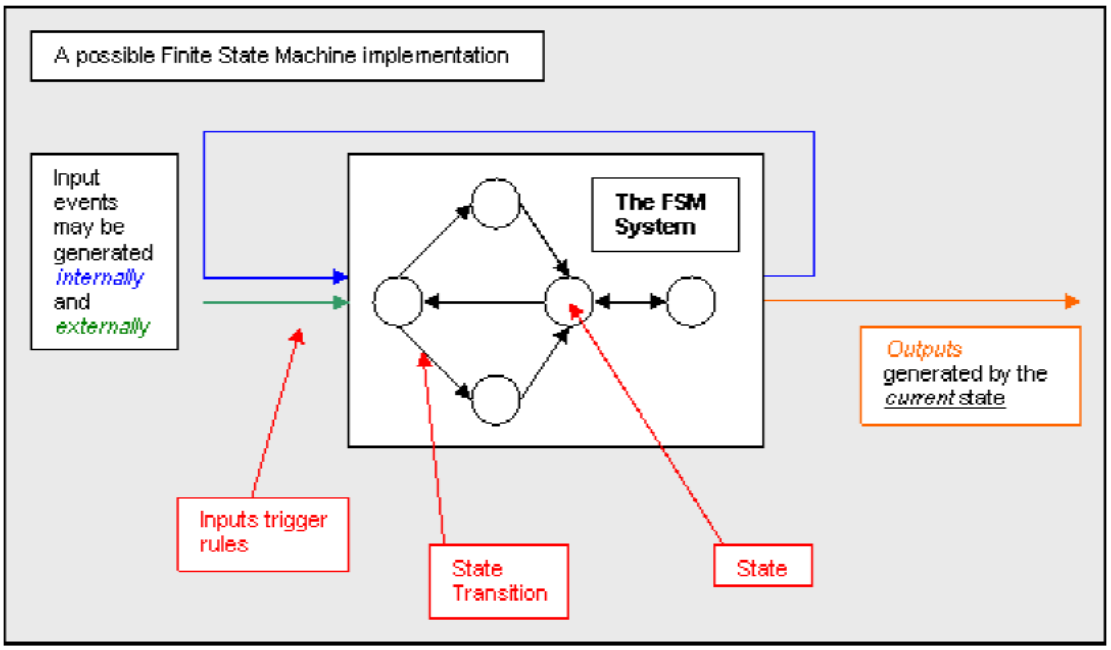


图 4

现在我们已经知道了有限状态机是什么了，那么我们来看看怎么在电子游戏中应用有限状态机。以下使用了有限状态机的电子游戏的例子都是来自同一家公司——id software 的第一个人称射击游戏 DOOM,Quake，Quake2。

(Matthews, 2000)提到了关于 Quake2 中机器人（npc 对手）行为的一个有趣的例子。我是用了他的 FSM 图来制作了图 5。

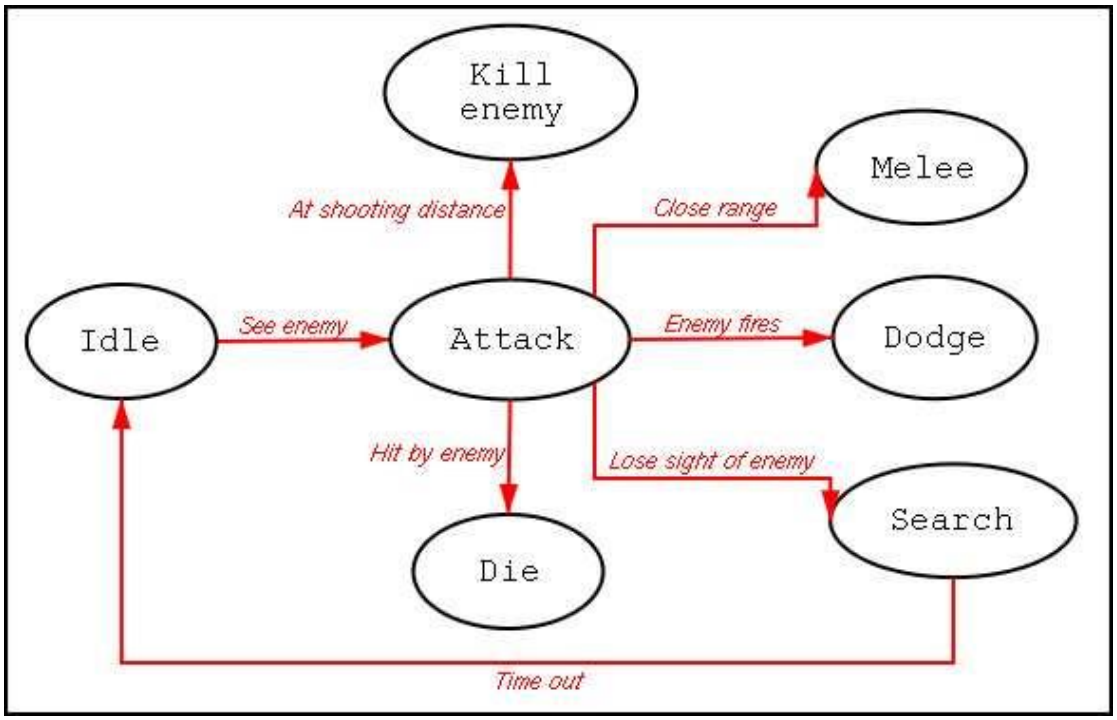


图 5

让我们注意机器人的行为。首先，机器人会处于空闲状态，什么也不做。当他看到敌人的时候，他会进入攻击状态。基于这个状态中机器人收到的输入事件，机器人会选择合适的下一步动作。尽管图 5 是一个简化的模型，但是他清晰的说明了 FSM 是怎么运作的。

(Matthews, 2000)中提到有限状态机是一个很好的方式来构建快速、简单、充足的游戏 AI 模型。当然，这也是有缺点的。(Brownlee,2002)评论说对于游戏，简单可预测的行为一般都不是想要的特征，因为趋向于移除游戏的快乐因子。他提到，使用可能性来拓展 FSM 可以使 NPC 的行为更加难以预测："一些扩展有限状态机，如随机选择过渡，以及模糊状态机显示我们另一种常见类型的 FSM 称为非确定性，系统动作不是可预测，能更好地呈现智能"。

3.4 总结

要开发电子游戏，您首先需要了解电子游戏的运作方式。你需要明白的第一件事是，一个电子游戏只是一个有 main 函数的程序。 Main 函数由五个状态组成。 电子游戏的三个最重要的状态是在游戏循环中彼此连接的，游戏循环是一系列获取输入和显示输出到玩家并更新游戏的过程。电子游戏的可玩性和人工智能是游戏循环中一个现代视频游戏必不可少的一些有趣的方面。但是，这一切都从方案开始。

方案应描述一个电子游戏中有可能发生的内容。 至少应该描述这个电子游戏的互动元素，这是游戏的基础。 而且，方案应该具有将交互元素彼此连接的情节。 现在大多数电子游戏有一个情节。 随着方案（部分）完成，可玩性可以开始设计。

可玩性决定了游戏怎么玩，可玩性受到一系列的因素影响：方案和游戏类型，控制与响应，视角（摄像机）以及学习曲线和游戏难度。由于可玩性是游戏的核心，他决定了玩家是否愿意继续玩下去。

人工智能（AI）是一系列模拟高级的只能来完成复杂任务的软件。在电子游戏中，人工智能被开发者用于给玩家的对手添加现实元素，提升游戏质量。这将会给玩家提供更真实、更好地游戏体验。AI 通常在玩家不能控制或者影响的游戏元素中应用。有不同种类的 AI 技术可以用于电子游戏之中。其中的三种为：搜索算法、神经网络和有限状态机。

四、如何开发 3D 游戏

在本章中，将会把前几章的知识放在一起，来弄明白如何制作视频游戏。 我们需要的第一个东西就是游戏引擎。(Simpson, 2002)通过比较明确地区分游戏和游戏引擎："许多人将引擎与整个游戏混淆。 那就像混合汽车发动机和整车一样。 您可以将发动机从汽车中取出，并在其周围建造另一个外壳，并再次使用。 游戏也是这样。 引擎可以被定义为所有非游戏特定技术（例如渲染器）。 游戏部分将是所有内容（模型，动画，声音，人工智能和物理），称为"资产"，以及专门用于使游戏工作的代码，如 AI，或控件如何工作。"

引擎在所谓的应用程序编程接口（API）之上构成。API 是一组可用于组件，应用程序或操作系统的功能。 对于个人电脑来说，它提供了一个前后不一致的前端（Simpson，2002）。 对于游戏机，前

端和后端都是一致的。 然而，对于这两种类型的系统，API 是必需的，因为 API 可以与（图形）硬件进行通话并进行必要的系统调用。

上述方法导致分层电子游戏结构（图 6）。 分层方法意味着层可以使用来自其的层的功能/可能性（较高层）。

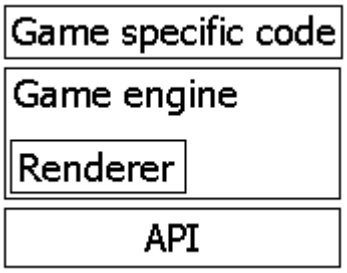


图 6

在下面的章节中，我将自底向上的讨论图 6 中的三层。

4.1 应用程序接口

如前所述，API 是一组用于访问较低级别服务的接口。可以使用的 API 取决于您要开发游戏的系统。对于游戏机，每个都有自己的专用 API，因为硬件是确定的。对于 PC 而言，API 需要更为通用，因为 PC 的硬件有很大差异。然而，您倾向于看到的是，当进行新的 PC 游戏时，它尽可能地利用最新的硬件，但同时也试图支持“旧”硬件。这将导致您的 PC 的最小配置应该与新的硬件技术可以并入多少之间取得平衡。目前 PC 上使用的两个主要 API 是 OpenGL 和 DirectX。 OpenGL 是一个纯图形导向的 API，其中 DirectX 是用于图形，输入处理，声音等的 API 的集合。 DirectX 与 OpenGL 对应的是 Direct3D。有关 OpenGL 或 Direct3D 哪个更好的讨论一直存在。在(Roy, 2002)对两者都进行了没有偏向的彻底的分析讨论。我同意(Roy, 2002)，您应该根据您的情况和平台选择某个 API（或两者）。最后，选择哪一个并不重要，因为一旦你学到了其中之一，学习其他的东西就不是太难了。

4.2 游戏引擎

游戏引擎是建立在 API 之上，由非游戏专用技术组成。给游戏引擎用第一个使用它的游戏的名称来命名是非常常见的。 一个经典的游戏引擎的例子就是 Quake 引擎。

游戏引擎与渲染器不同。 渲染器仅构成游戏引擎的图形部分。 引擎可能不仅仅是一个渲染器，它可以包括世界编辑器，字符编辑器，重力模型和碰撞模型。 引擎不包含包含最大值。 这些例子也可以省略; 这只是你想要的东西的一个问题。 请记住，如果您希望将引擎重新用于其他游戏，则（最好）不应该包含任何特定于游戏的技术。

如前所述，渲染器是游戏引擎的图形组件。 它是游戏引擎最少应该包含。(Simpson, 2002) 给出了渲染器做什么的好定义：“渲染器可以使玩家/观众的场景可视化，以便他或她可以根据所显示的内容做出适当的决策。 在构造引擎时，通常是您倾向于构建的第一件事。 渲染器是花费了超过 50％的处理器时间，而游戏开发人员经常被要求最苛刻。 今天在屏幕上获取像素的业务涉及 3D 加速卡，API，

三维数学，3D 硬件如何工作的理解以及一些魔法灰尘。 对于游戏机来说，需要同样的知识，但至少在游戏机上，你不是想要达成目标。 游戏机的硬件配置是冻结的“时间快照”，与 PC 不同，游戏机的整个生命周期都不会改变。

在一般意义上说，渲染者的工作就是创建一个使游戏与牛群分开的视觉耀斑，实际上拉开这个需要巨大的创造力。 3D 图形本质上是创造最多的艺术，同时做到最少，因为在处理器周期和存储器带宽方面，额外的 3D 处理通常都是昂贵的。 这也是一个预算问题，弄清楚你想在哪里度过周期，以及你愿意为了达到最佳整体效果而舍弃的角色。 “有关游戏引擎的更多信息，我推荐前面提到的教程（Simpson，2002）。

4.3 游戏实现代码

游戏特定代码基本上由前面章节描述的游戏循环组成，其中包含构成游戏和 AI 的交互元素。 此外，您不能放入游戏引擎的所有内容都是游戏特定代码的一部分。 你可以想到具体的角色模型，声音，特殊的碰撞模型等等。 只要它不包含在引擎中，您可以将其添加到此层。 通常放在游戏特定代码中的其他功能是多人/网络选项和 HUD /菜单。

当您使用现有的引擎进行视频游戏时，您已经有了一个很好的起点。然而，您希望为引擎添加更多功能并不是不可想象的，因为并不总是将它放在游戏中的特定代码中。这取决于游戏引擎的组合方式，可以扩展它并添加所需的额外功能。

最后我想讨论游戏编辑器的使用。 通过编码创建整个世界是不切实际的。 现在很多游戏都使用了特制的编辑器。 这些编辑器允许创建游戏世界，放置角色和对象，脚本等等。 在游戏发布时，电子游戏包含编辑器（修改后的形式）并不罕见。 这给玩家机会，建立自己的世界玩，从而延长视频游戏的使用寿命。 进一步的是所谓的 mod（ification），它允许修改游戏特定的代码，有时甚至是（引擎的）部分。 一个非常知名的 mod 是 Counterstrike，它是基于 Half-Life 引擎制作的。 它甚至被发布为一个新的（单独的）视频游戏。

4.4 总结

在制作电子游戏时，会采用分层的方法（图 6）。 底层是 API，游戏引擎和游戏专用代码。

API 是一组用于访问较低级别服务的接口。 可以使用的 API 取决于您要开发游戏的系统。 对于游戏机，每个都有自己的专用 API，因为硬件没有改变。 对于 PC 而言，API 需要更为通用，因为 PC 的硬件有很大差异。 目前使用的两大 PC API 是 OpenGL 和 DirectX。

游戏引擎建立在 API 之上，由非游戏特定技术组成。游戏引擎与渲染器不同。 渲染器仅构成游戏引擎的图形部分。 发动机应该包含的最大值。 如果你想让你的引擎重新用于其他游戏，它（最好）不应该包含任何游戏特定的技术。

渲染器所做的是可视化播放器/查看器的场景，以便他或她可以根据所显示的内容做出适当的决定。在构造引擎时，通常是您倾向于构建的第一件事。

游戏特定代码由游戏循环组成，其中包含组成游戏和 AI 的交互元素。 此外，您不能放入游戏引擎的所有内容都是游戏特定代码的一部分。 通常放在游戏特定代码中的其他功能是多人/网络选项和 HUD /菜单。

当使用现有的引擎时，可能会扩展它以增加你的游戏需要的额外的功能。

游戏编辑器允许创建游戏世界，放置角色和对象，脚本等等。 在游戏发布时，电子游戏包含游戏编辑器（修改后的形式）并不罕见。 这给玩家创造自己的世界玩的机会。进一步的是一个所谓的 mod（ification），它允许修改游戏特定的代码，有时甚至是（引擎）的部分。

五、总结

在介绍中我提出以下问题：

- 制作 3D 游戏需要什么？
  - 什么是 3D 图形的基本概念？
  - 什么是电子游戏的基本概念？
  - 您如何将前面的问题获得的知识组合在一起，制作 3D 视频游戏？

现在，我将对每一个子问题给出答案：

- 什么是 3D 图形的基本概念？

首先，需要定义一个 3D 坐标系。 其次，您需要选择如何在坐标系中表示对象。 您可以从不同种类的对象模型中进行选择。 目前，多边形（mesh）模型是事实上的标准，但随着 CPU 和更多内存的可用性的提高，这一变化正在改变。 最后，您需要一种方法来操纵坐标系中的对象。 这可以用仿射变换来完成。 计算机图形的三个最基本和常用的仿射变换是平移，旋转和缩放。 可以通过应用正确选择的上述基本变换序列来创建任何其他仿射变换。

- 什么是电子游戏的基本概念？

你需要了解的第一件事是，一个电子游戏只是一个由 main 函数开始的程序。 main 函数包括五个状态。 电子游戏的三个最重要的状态在游戏循环中彼此连接，这是一系列用于获取输入并向玩家显示输出并更新游戏的过程。 来自游戏循环的一些有趣的方面如游戏的可玩性和人工智能对于现代视频游戏至关重要。 但是，这一切都从游戏方案开始。

该方案应描述一个视频游戏中可能的内容。 至少应该描述电子游戏的互动元素，这是可玩性的基础。 此外，该方案可以具有将交互元素彼此连接的情节。

可玩性确定游戏的玩法。可玩性游戏受到以下几个因素的影响：场景和流派，控制和反应，视角（相机）和学习曲线以及易玩性。 因为可玩性是每个游戏的核心，它将决定玩家是否想玩它。

人工智能（AI）是软件的通用术语，用于模拟高级智能以执行复杂任务。 在电子游戏中，AI 被游戏开发人员用来添加一个现实主义的元素来挑战一个人的对手，提高游戏的质量。 这应该为

玩家提供更好更实际的体验。 AI 通常应用于玩家不能控制或影响的游戏世界的元素。 有不同种类的 AI 技术可用于电子游戏。

• 您如何将前面的问题获得的知识组合在一起，制作 3D 视频游戏？

为了制作 3D 视频游戏，采用分层方法（图 6）。 底层是 API，游戏引擎和游戏专用代码。

API 是一组用于访问较低级别服务的接口。 可以使用的 API 取决于您要开发游戏的系统。

游戏引擎是建立在 API 之上，由非游戏专用技术组成。 游戏引擎与渲染器不同。 渲染器仅构成游戏引擎的图形部分。 发动机应该包含的最大值。 如果你想让你的引擎重新用于其他游戏，它（最好）不应该包含任何游戏特定的技术。

渲染器所做的是可视化播放器/查看器的场景，以便他或她可以根据所显示的内容做出适当的决定。

游戏特定代码由游戏循环组成，其中包含组成游戏和 AI 的交互元素。 此外，您不能放入游戏引擎的所有内容都是游戏特定代码的一部分。

游戏编辑器允许创建游戏世界，放置角色和对象，脚本等等。 在游戏发布时，电子游戏包含游戏编辑器（修改后的形式）并不罕见。 这给玩家创造自己的世界玩的机会。进一步的是一个所谓的 mod（ification），它允许修改游戏特定的代码，有时甚至是（引擎）的部分。

然后是主问题的答案：

• 制作 3D 游戏需要什么？

电子游戏的分层结构说明它是如何构建的，因此你需要知道您将需要通过获取（阅读：购买或构建）游戏引擎来开始。 此外，您将始终需要 3D 图形的知识，因为在游戏引擎（渲染器）中，您将始终需要处理它。 最后，你必须了解视频游戏的运作方式。 否则你不能开发方案，可玩性，AI 等等。

上面的答案当然不是一切都是为了制作一个电子游戏。 构建电子游戏也需要很多经验。 但是，当我从理论的角度来看，就像我所做的一样，我认为我在本文中描述的是有意义的。

附录二 文献原文

Abstract

Video games have become a common occurrence in today's world. In this paper I have looked at what makes 3D gaming possible. To understand 3D video games, the basics of 3D graphics are the starting point. To make a video game, you need to understand how a video game works.When making a video game, a layered approach is followed. The layered structure of a video game tells how it is built up and thus what you need to know to make one.

Introduction

Video games have become a common occurrence in today's world. Just like other software, it has evolved over the years. For video games the biggest and most noticeable change has been made graphics wise. The first video games were in 2D i.e. in a two-dimensional plane.Today's video games mostly are in 3D. The step from 2D to 3D graphics was made possible by technological progress in the computer industry. In this paper I want to look at what makes 3D gaming possible. Therefore I have the following questions:

·What does it take to make a 3D video game?

· What are basic concepts for 3D graphics?

· What are basic concepts for a video game?

·How do you put together the knowledge gained from the previous questions to make a 3D video game?

I will devote a chapter per sub question. In the conclusion I will summarize the discussed material and give answers to the above questions.

2.Basic concepts of 3D graphics

According to (Watt et al, 2001) the role of three-dimensional computer graphics in the context of computer games is to supply tools that enable the building of worlds, populating them with characters and objects, having these games objects interact with each other and 'reducing' the action to a two-dimensional screen projection in real time. None of this can be accomplished without knowledge of the basic mathematics of three-dimensional space. In this chapter I will talk about these basic mathematics, but I will not go into gory mathematical details. Instead, I will give a basic overview that serves as an introduction towards understanding 3D graphics in the context of 3D video games.

2.1. 3D coordinate system

It all starts with a three-dimensional coordinate system. This system has three axes: x, y and z.There are two common ways to arrange the (positive) axes: the right-handed and the left-handed system (figure 1 and 2).

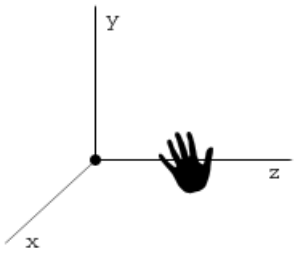


Figure 1 Left-handed coordinate system

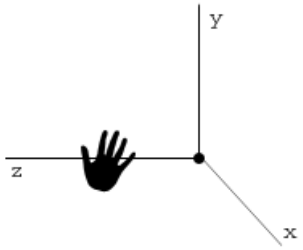


Figure 2 Right-handed coordinate system

To determine with which system you are dealing, visualize wrapping one of your hands around the z-axis. Then look at the direction your thumb is pointing. If it points in the same direction, then the hand you used is also the name for the system. If not, the name for the system is the name of the other hand.

Although right-handed systems are the standard mathematical convention, both systems are used. The difference between the two systems is the position the (positive) z-axis points to.

2.2. Objects

With the coordinate system in place, it is time to move on to the next topic: objects. At their lowest level, three-dimensional objects are represented as a set of points in the coordinate system. For simple objects you could define points in the three-dimensional space and draw lines between them to form the object. For complex objects this would be a tedious job; therefore you need a different way to represent an object in 3D. In (Watt et al, 2001) a number of mainstream computer graphics models are mentioned:

1. Polygonal model.

   Objects are approximated by a net or mesh of planar polygonal facets (polygons). In (Angel, 2003) a polygon is described as an object that has a border that can be described by a line loop, but has an interior. That is also the reason that it is sometimes referred to as fill area. With polygons you can represent objects to an accuracy of choice. More accuracy comes at the cost of using more polygons.

2. Bi-cubic parametric patches.

   These are ‘curved quadrilaterals’. Generally it can be said that the representation is similar to the polygon mesh except that the individual polygons are now curved surfaces. Each patch is specified by a mathematical formula that gives the position of the patch in three-dimensional space and its shape. This formula enables us to generate any or every point on the surface of the patch. The shape or curvature of the patch can be changed by editing the mathematical specification. This results in powerful interactive possibilities. The problems are, however, significant. When the shape of individual patches in a net of patches is changed, there are problems in maintaining ‘smoothness’ between the patch and

its neighbors. Bi-cubic parametric patches can either be an exact or an approximate representation. They can only be an exact representation of themselves, which means that any object can only be represented exactly if the shape corresponds exactly to the shape of the patch. This somewhat torturous statement is necessary because when the representation is used for real or existing objects, the shape modeled will not necessarily correspond to the surface of the object.

A significant advantage of the representation is its economy. When compared to the polygonal model, the bi-cubic parametric patches model uses fewer elements to create the same object.

3. CSG (constructive solid geometry).

This is an exact representation to model objects within certain rigid shape limits. It has arisen out of the realisation that a lot of manufactured objects can be represented by 'combinations' of elementary shapes or geometric primitives. For example, a chunk of metal with a hole in it could be specified as the result of a three-dimensional subtraction between a rectangular solid and a cylinder. Connected with this is the fact that such a representation makes for easy and intuitive shape control – it can be specified that a metal plate has to have a hole in it by defining a cylinder of appropriate radius and subtracting it from the rectangular solid, representing the plate. The CSG method is a volumetric representation: shape is represented by elementary volumes or primitives. This contrasts with the previous two methods that represent objects using surfaces.

4. Spatial subdivision techniques.

With this technique the object space is divided into elementary cubes, known as voxels and each voxel is labelled as empty or as containing part of an object. It is the threedimensional analogue of representing a two-dimensional object as the collection of pixels onto which the objects projects. Labelling all of three-dimensional object space in this way is clearly expensive, but it has found applications in computer graphics. This model thus represents the three-dimensional space occupied by the object; the first two mentioned methods are representations of the surface of the object.

5. Implicit representation.

Implicit functions are occasionally mentioned in texts as an object representation form. An implicit function is, for example:

$$\mathbf{x^2 + y^2 + z^2 = r^2}$$

which is the definition for a sphere. On their own they are of limited usefulness in computer graphics because there are a limited number of objects that can be represented in this way. Also, it is an inconvenient form as far as rendering is concerned. However, it should be mentioned that such representations do appear quite frequently in three-dimensional computer graphics – in particular in

ray tracing where spheres are used frequently – both as objects in their own right and as bounding objects for other polygon mesh representations. Implicit representations are extended into implicit functions that can loosely be described as objects formed by mathematically defining a surface that is influenced by a collection of underlying primitives such as spheres. Implicit functions find their main use in shape-changing animation; they are of limited usefulness for representing real objects. The method does show potential for modelling organic shapes. Metaballs (blobs) is an implicit modelling technique that is used for creating the aforementioned organic shapes.

From all the object models, the polygonal model is the de facto standard. The main reason is that it has low CPU requirements; however you have the accuracy/polygon count trade-off. Other object models like bi-cubic parametric patches and spatial subdivision techniques (voxels) are getting used more frequently for 3D games. Although these models have higher CPU requirements, their advantages over polygons are enough reasons to start using them. The emergence of faster CPUs and more available memory will also stimulate the usage of other available models.

2.3. Transformations

In (Watt et al, 2001) transformations are described as important tools in generating threedimensional scenes. They are used to move objects around in an environment, and also to construct a two-dimensional view of the environment for a display surface. The basic idea behind a transformation is mapping a point to another point using some kind of function. The possible functions correspond to the different kinds of transformations. The transformations used for computer graphics are called affine transformations. According to (Weisstein, 2004), an affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation). In this sense, affine indicates a special class of projective transformations that do not move any objects from the affine space to the plane at infinity or conversely. From a computer graphics perspective this means that an object can still be represented in the user defined 3D coordinate system after a transformation.

Affine transformations can be represented by a matrix and a set of affine transformations can be combined into a single overall affine transformation by combining the separate matrices. This makes it possible to apply different kinds of transformations to a single object. For more information on the mathematical background of affine transformations and matrices, see (Angel, 2003).

The most basic and commonly used affine transformations in computer graphics are translation, rotation and scaling. In (Angel, 2003) shear is also presented as a basic affine transformation, because of its importance. However, since it can be created by applying a sequence of the aforementioned three basic affine transformations,

I do not see it a basic affine transformation. I will now give short descriptions of translation, rotation and scaling. These are abbreviations from (Angel, 2003).

Translation is an operation that displaces points by a fixed distance in a given direction. Since we are dealing with a 3D coordinate system, the translations take place in the three axis directions.

Rotation is in an operation that displaces a fixed point around a certain vector at a certain angle. For a given fixed point there are three degrees of freedom: the two angles necessary to specify the orientation of the vector and the angle that specifies the amount of rotation about the vector.

Scaling makes objects bigger or smaller in any (combination) of the three dimensions. Scaling transformations have a fixed point. To specify a scaling, you need to specify the fixed point, a direction in which you want to scale and a scale factor.

2.4. Summary of basic 3D graphics concepts

To understand 3D videogames, the basics of 3D graphics are the starting point. Firstly, a 3D coordinate system needs to be defined. Secondly, you need to choose how to represent you objects in your coordinate system. You can choose from different kinds of object models, each with their trade-offs. Currently the polygon (mesh) model is the de facto standard, but this is changing as faster CPU's and more memory become available. Lastly, you need a way to manipulate the objects in your coordinate system. This can be done with affine transformations. The three most basic and commonly used affine transformations for computer graphics are translation, rotation and scaling. Any other affine transformation can be created by applying a properly chosen sequence of the aforementioned basic transformations.

3.Basic concepts of video games

To make a video game, you first need to understand how a video game works. The first thing you need to realise is that a video game is just a program that can be compiled and executed. Just like any other program it has a main function that is the first function to be called upon when it is executed. The structure of the main function can be represented as a flowchart diagram, as depicted in figure 3 (Kuffner, 2002).

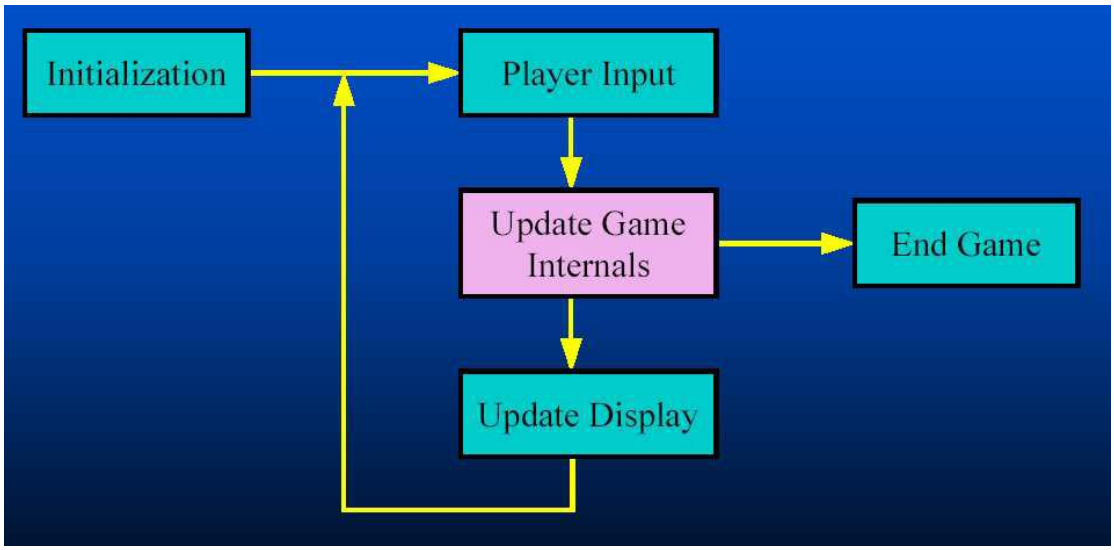


Figure 3 Flowchart diagram of the main function of a videogame

The diagram above consists of five states. The three most important states for a video game are connected to each other in a loop. This loop is often referred to as the "game loop". In (Howland, 2001) the game loop is described as a series of procedures for getting input and displaying output to the player and updating the game. Let us now look at each state and their meaning:

·Initialization (Starting the game)

According to (Howland, 2001) the initialization is as follows: "The beginning of every game usually consists of an animation sequence to show off some aspect of the game's story or background and an option menu for starting the game or changing various parameters that affect the game in some way. Those parameters often include sound volume, graphic options, multiplayer options, difficulty and starting level." I agree that this is the way a videogame is started when you play it. But when think about it, these things are actually done in the so-called game loop, because the player inputs actions and the videogame responds by performing the appropriate action. The changes have to be updated internally and have to be outputted to the player. What I think is meant with initialization is that the necessary parameters are set, functions are called et cetera that are necessary so that the videogame can move from this state to the game loop. Examples of what could be done in the initialization state are checking and requesting memory, setting up a stack, load display drivers.

·Player input

(Howland, 2001) says the following: "The input gathering routines will take the player's input from whatever device they are using and store it in a way that the game can process it to make changes to the game internals." Note that the stored input information is used in the next state (updating game internals).

·Updating game internals

Here is what (Howland, 2001) has to say: "The game updating routines are the real guts of the game. Everything from moving the player's character using their input, to the actions of the enemies

and determining whether the player has won or lost the game is determined here." Also, preparations are made for displaying the graphics.

·Displaying the screen

According to (Howland, 2001) the screen can be displayed in two ways: "You can either draw everything to the screen at one time or what is more commonly done, set everything up to be drawn and then draw the screen afterwards. Drawing the screen can take longer than most processes in a game so you want to do it all at one time. Determining what is necessary to be drawn can take a relatively long time with all the checks your game could have, so it's best to do this before you actually try to commit your graphics to the screen or video hardware."、

·Ending the game

(Howland, 2001) describes that normally when a game ends, an ending sequence is displayed, or at least something that says goodbye to the player. This description is in the same spirit as his description about starting a game: it describes this state from a player's point of view. Again, I think that something else is meant, namely calling functions and setting values et cetera, so that the video game can be terminated in a normal way.

Now that I have given an overview of the how a game basically works, it is time to look at some interesting aspects from the game loop that are essential for a modern video game. I will focus on the updating of the game internals and more specifically on gameplay and artificial intelligence of a video game. Before I start with discussing these subjects, I would first like to take a look at the scenario of a video game.

3.1. Scenario

According to Webster's dictionary a scenario is a preliminary sketch of the plot, or main incidents. In the context of video games the word scenario has the same meaning, but there is one more thing that should be taken into consideration: a video game is interactive. When making the scenario for a game, it is quite usual nowadays to have some kind of plot that connects the interactive elements of the video game. Even when there is no (real) story, the scenario should describe what is possible in a video game. This leads to the following view of what the scenario of a video game looks like:

·The scenario at least describes the interactive elements of the video games. This should be written down as much as possible in a non-technical way, because you are not programming (yet). These interactive elements determine the gameplay.

·The scenario can have a plot that connects the interactive elements to each other.This part of the scenario can be seen in the same way as a scenario from a movie or a play. Nowadays most video games have a plot. For some genres of games this can lead to quite ridiculous plots, for instance puzzle games. Other genres are very dependant of the plot and cannot exist without it. In this case the plot is equally and sometimes more important than the interaction. An example is the genre of role-playing games. In this case

you will need a system to present the story to the players. There are various ways for doing this, as described in (Simpson, 2002). You could for instance use cut scenes or scripting.

With the scenario (partially) developed, the gameplay can be worked out further.You can do this by writing down more about it, or [when worked out enough] you can start with programming experiments to test it out.

3.2. Gameplay

The word gameplay is one of the keywords in video gaming today. As the word says, it determines how a game is played. Gameplay is influenced by a couple of factors:

·Scenario and genre

In the scenario at least the interactive elements of a video games are written down,which determine the basic gameplay form. What has not been mentioned earlier is choosing the right genre(s) for a game. Each genre brings along certain gameplay elements that may or may not suit the game scenario. It is possible to combine different elements from different genres to fit the scenario. So it is the combination of scenario and genre(s) and how it is worked out that determine the basic gameplay.

·Control and response

Very important refinements of the gameplay are how a game is controlled and how well it responds to player input.The game controls should be defined according to what the player expects and knows and/or should be as natural as possible. Also, for advanced players customizable controls are recommended. A well thought up control scheme can benefit the gameplay, because it lets the game be played more easily. Controlling a game bring up the issue of response. The player should always be able to see what the consequences of their actions are in the video game. The response time should always be as short as possible, according to the type of gameplay. For instance, in a race game you do not want a response time of 5 seconds when racing, because in that time span, something else has already happened.

·Viewpoint (camera)

Although the viewpoint does not affect the gameplay directly, its choice does influence the way the game is played. If the player has a wrong view of the game world, it is harder to make a decision. For 3D games the viewpoint is often referred as the camera. The idea behind this is that the camera follows the player to show the gaming world.

·Learning curve and ease of play

The last two factors of influence for gameplay are the learning curve and ease of play. I have placed these two together because they both deal with the level of difficulty in a game.

The learning curve is the initial difficulty encountered when first playing a game. This affects the gameplay in such a way that only if the player knows the basics about the game, he/she can effectively

play it. Thus if the player do not know enough, the gameplay is reduced, since certain gameplay elements are missed. The ease of play is the difficulty you engage in the game after the learning curve. The player at least knows how the basic gameplay elements work. This does not mean that the player has mastered a game. In many games it is possible to set the difficulty. This usually affects how the game world responds to the player. In the next paragraph about artificial intelligence I will come back on this.

Because gameplay is the core of each game, it will determine if the player wants to play it or not. Where graphics are dependant of technology and show their age over time, well thought out gameplay can last forever.

3.3. Artificial intelligence

According to (Kelly, 2003), Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In video games AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game (Kelly, 2003). This should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. A so-called non-playable character (NPC) is an example of this. There are different kinds of AI techniques available that can be used for video games. Three of these [commonly used] techniques are search algorithms, neural networks and finite state machines. In the following paragraphs I will discuss their uses in video games.

3.3.1. Search algorithms

Search algorithms can be used to solve path-finding problems. An example of a path-finding problem in a video game is an opponent who tries to locate the player.

(Kelly, 2003) discusses search algorithms as having a problem space model, which is the environment where the search is performed. This consists of a set of states of the problem and a set of operators that can alter the state. Although brute force search algorithms like depth-first and breath-first search always find their goal, they are usually not suitable for video games, since they can require lots of memory and are inefficient as they can explore a large number of possible states until the goal state is reached. The A* (A star) algorithm is also a path finding algorithm, but unlike the brute force algorithms it is more efficient. For more information about A*, see (Patel, 2003). Another search algorithm is the minimax algorithm. It looks ahead to try and anticipate what the opponent will do. This information is used to choose the best move that gives the computer the advantage (Kelly, 2003).

Whichever search algorithm is used depends on the video game. The trade-off that has to be made is speed versus accuracy. For most games, you don't really need the best path between two points. You just need something that is close. What you need may depend on what is going on in the game, or how fast the computer is.

(Patel, 2003) talks about this trade-off in the context of A*, but it can be generalized to search algorithms in general.

3.3.2. Neural networks

When talking about neural networks or neural nets (NN) the first question of course is: what is a neural network? (Sarle et. al., 1997) gives us the following answer: “There is no universally accepted definition of an NN. But perhaps most people in the field would agree that an NN is a network of many simple processors (‘units’), each possibly having a small amount of local memory. The units are connected by communication channels (‘connections’), which usually carry numeric (as opposed to symbolic) data, encoded by any of various means. The units operate only on their local data and on the inputs they receive via the connections. The restriction to local operations is often relaxed during training.”This definition brings up an interesting aspect that makes NN interesting for use in video games: its ability to learn. (Kelly, 2003) justly remarks that this feature makes them an interesting addition to any game as they can gradually adjust themselves to provide a more challenging experience for the player. About the training rules (Sarle et. al., 1997) has the following to say: “Most NNs have some sort of ‘training’ rule whereby the weights of connections are adjusted on the basis of data. In other words, NNs "learn" from examples, as children learn to distinguish dogs from cats based on examples of dogs and cats. If trained carefully, NNs may exhibit some capability for generalization beyond the training data, that is, to produce approximately correct results for new cases that were not used for training.”

(LaMothe, 1999) gives a couple of examples of NN video game applications:

·Environmental scanning and classification.

A neural net can be feed with information that could be interpreted as vision or auditory information. This information can then be used to select an output response or teach the net. These responses can be learned in real-time and updated to optimize the response.

·Memory.

A neural net can be used by game creatures as a form of memory. The neural net can learn through experience a set of responses. Then when a new experience occurs, the net can respond with something that is the best guess at what should be done.

·Behavioral control.

The output of a neural net can be used to control the actions of a game creature. The inputs can be various variables in the game engine. The net can then control the behavior of the creature.

·Response mapping.

Neural nets are really good at ‘association’, which is the mapping of one space to another. Association comes in two flavors: autoassociation, which is the mapping of an input with itself and heterassociation, which is the mapping of an input with something else. Response mapping uses a neural net at the back end or output to create another layer of indirection in the control or

behavior of an object. Basically, we might have a number of control variables, but we only have crisp responses for a number of certain combinations that we can teach the net with. However, using a neural net on the output, we can obtain other responses that are in the same ballpark as our well defined ones.

(LaMothe, 1999) remarks that the above examples may seem a little fuzzy, and they are. The point is that neural nets are tools that we can use in whatever way we like. The key is to use them in cool ways that make our AI programming simpler and make game creatures respond more intelligently.

3.3.3. Finite state machines

In (Brownlee, 2002) finite state machines are described as follows: ”Finite state machines (FSM),also known as finite state automation (FSA), at their simplest, are models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes, where mode transitions change with circumstance. Finite state machines consist of 4 main elements:

·States, which define behavior and may produce actions

·State transitions, which are movement from one state to another

·Rules or conditions, which must be met to allow a state transition

·Input events, which are either externally or internally generated, which may possibly trigger rules and lead to state transition

A finite state machine must have an initial state, which provides a starting point, and a current state, which remembers the product of the last state transition. Received input events act as triggers, which cause an evaluation of some kind of the rules that govern the transitions from the current state to other states. The best way to visualize a FSM is to think of it as a flow chart or a directed graph of states, though there are more accurate abstract modeling techniques that can be used.” Figure 4 shows a picture of a FSM by (Brownlee, 2002).

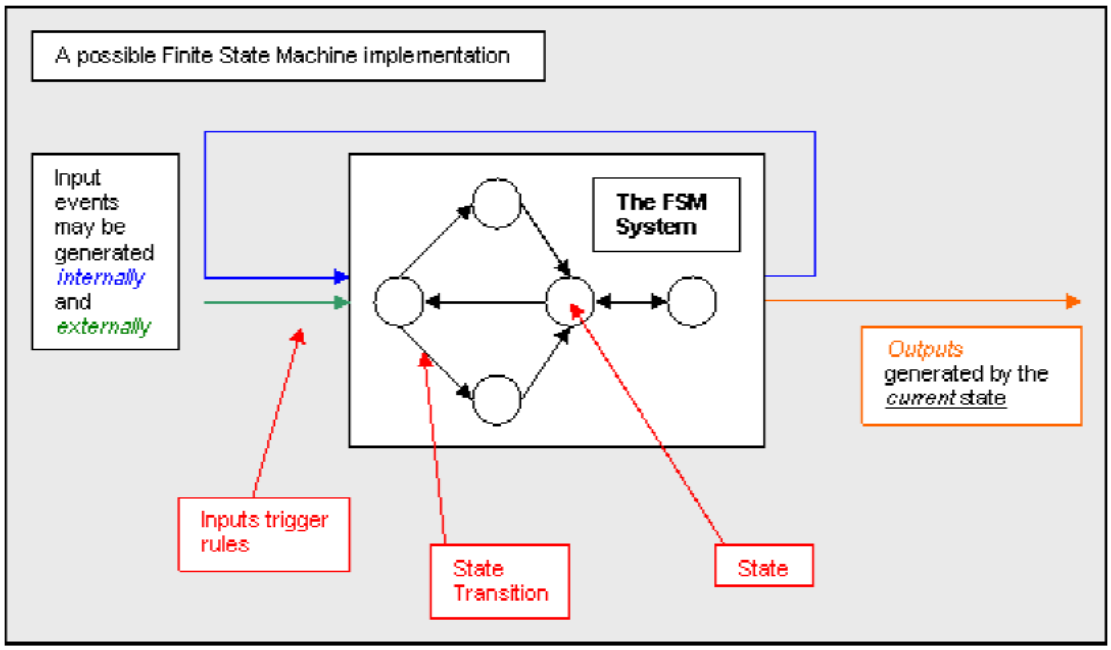


Figure 4 A possible finite state machine control system implementation

Now that we know what a FSM is, let us look at how we can use it for a video game.A few examples of videogames that use FSMs are the first person shooter (FPS) games Doom, Quake and Quake 2, which were all made by the same company, id Software. (Matthews, 2000) has an interesting example dealing with how the bots (NPC opponents) behave in his Quake 2 mod(ication). I have used his original FSM diagram to make figure 5.

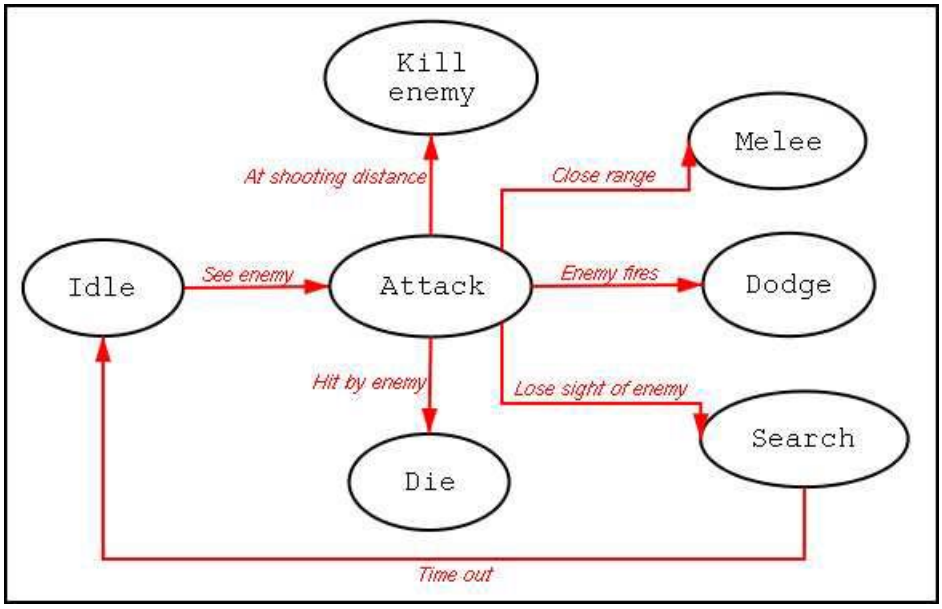


Figure 5 Finite state machine of a Quake 2 mod

Let us take a look at the behavior of the bot. Firstly the bot will be in the idle state, doing nothing. When he sees the enemy, it will go to the attack state. Depending input events the bot receives while in this state, the bot will choose the appropriate follow-up action. Although figure 5 is a simplified model, it does make clear how a FSM works.

Finite-state machines are a good way to create a quick, simple, and sufficient AI model for the games it is incorporated in (Matthews, 2000). This also has its disadvantages. (Brownlee, 2002) justly remarks that for computer games, easily predictable behavior is usually not a wanted feature, as it tends to remove the 'fun-factor' in the game. He mentions possibilities to extend the FSM to make it harder to predict (NPC) actions: "A number of extensions to finite state machines such as random selection of transitions, and fuzzy state machines shows us another common type of FSM called non-determinist where the systems actions were not as predictable, giving a better appearance of intelligence."

3.4. Summary of basic video game concepts

To make a video game, you first need to understand how a video game works. The first thing you need to understand is that a video game is just a program with a main function. The main function consists of five states. The three most important states for a video game are connected to each other in the game loop, which is a series of procedures for getting input and displaying output to the player and updating the game. Some interesting aspects from the game loop that are essential for a modern video game are gameplay and artificial intelligence of a video game. However, it all starts with the scenario.

The scenario should describe what is possible in a video game. It should at least describe the interactive elements of the video games, which is the base for the gameplay. Furthermore, the scenario can have a plot that connects the interactive elements to each other. Nowadays most video games have a plot. With the scenario (partially) developed, the gameplay can be worked out further.

Gameplay determines how a game is played. Gameplay is influenced by a couple of factors: scenario and genre, control and response, viewpoint (camera) and the learning curve and ease of play. Because gameplay is the core of each game, it will determine if the player wants to play it or not.

Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In video games AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game. This should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. There are different kinds of AI techniques available that can be used for video games. Three of these techniques are search algorithms, neural networks and finite state machines.

4.Making a 3D video game

In this chapter the knowledge from the previous chapters is put together to prescribe how to make a video game. The first thing we need is a game engine. (Simpson, 2002) makes a clear distinction between the game and game engine by making a comparison: "Many people confuse the engine with the entire game. That would be like confusing an automobile engine with an entire car. You can take the engine out of the car, and build another shell around it, and use it again. Games are like that too. The engine can be defined as all the non-game

specific technology (for instance the renderer). The game part would be all the content (models, animations, sounds, AI, and physics), which are called 'assets', and the code required specifically to make that game work, like the AI, or how the controls work."

The engine on its turn is made on top of a so-called application programming interface (API). An API is a set of functions you can use to work with a component, application, or operating system. For PCs, it provides a consistent front end to an inconsistent back end (Simpson, 2002). For game consoles both the front and back end are consistent. Nevertheless for both types of systems an API is necessary, because an API can 'talk' to the (graphical) hardware and make the necessary system calls.

The above described approach results in a layered video game structure (figure 6). A layered approach means that a layer can use functionalities/possibilities from the underlying layer that are presented to it (the higher layer).

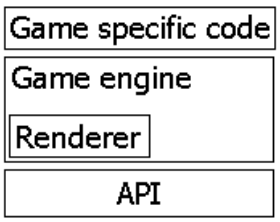


Figure 6 Layered structure of a video game

In the following paragraphs I will discuss the three layers of figure 6 bottom-up.

4.1. Application programming interface (API)

Like stated earlier, an API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for. For game consoles each has its own specialized API(s), since the hardware does not change. For PCs the APIs need to be more general, since there is a lot of difference between the hardware of PCs. However, what you tend to see is that when a new PC game is made, it tries to take advantage of the latest hardware as possible, but at the same time it tries to support 'older' hardware as well. This results in a trade-off between what the minimal configuration of your PC should be versus how much of the new hardware technology can be incorporated. Currently the two major APIs used on the PC are OpenGL and DirectX. OpenGL is a pure graphics oriented API, where as DirectX is a collection of APIs for graphics, input handling, sound et cetera. The OpenGL counterpart of DirectX is Direct3D. There is an ongoing discussion on whether OpenGL or Direct3D is better. In (Roy, 2002) both are discussed through a thorough analysis without favoring one. I agree with (Roy, 2002) that you should choose for a certain API (or both!) depending on your situation and platform. In the end it does not matter which one you choose, because once you have learnt one of them, it is should not be too hard to learn the other.

4.2. Game engine

The game engine is built on top of an API and consists of the non-game specific technology. It is very common to give a game engine the name of the first game that used it. A classic example of a game engine is the Quake engine.

The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. An engine can consist of much more than just a renderer, for instance it can have world editors, character editors, a gravity model and a collision model. There is no maximum to what an engine should contain. The examples could also be left out; it is just a matter of what you want to be in it. Keep in mind that if you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.

As mentioned earlier the renderer is the graphical component of a game engine. It is the minimum a game engine should contain. (Simpson, 2002) gives a good impression of what the renderer does: “The renderer visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed. It is generally the first thing you tend to build when constructing an engine. The renderer is where over 50% of the CPUs processing time is spent, and where game developers will often be judged the most harshly. The business of getting pixels on screen these days involves 3D accelerator cards, APIs, three-dimensional math, an understanding of how 3D hardware works, and a dash of magic dust. For consoles, the same kind of knowledge is required, but at least with consoles you are not trying to hit a moving target. A console's hardware configuration is a frozen ‘snapshot in time’, and unlike the PC, it does not change at all over the lifetime of the console.

In a general sense, the renderer's job is to create the visual flare that will make a game stand apart from the herd, and actually pulling this off requires a tremendous amount of ingenuity. 3D graphics is essentially the art of the creating the most while doing the least, since additional 3D processing is often expensive both in terms of processor cycles and memory bandwidth. It is also a matter of budgeting, figuring out where you want to spend cycles, and where you're willing to cut corners in order to achieve the best overall effect. “ For more information about how a game engine looks like, I recommend the tutorial by the earlier mentioned” (Simpson, 2002).

4.3. Game specific code

The game specific code basically consists of the in the previous chapter described game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code. You can think of specific character models, sounds, special collision models et cetera. As long as it is not contained in the engine, you can add it to this layer. Other features that are usually put in the game specific code are multiplayer/networking options and HUDs/menus.

When using an existing engine for your videogame you already have a good starting point. However, it is not unthinkable that you would like to add more functionality to the engine, because it is not always possible to put that in the game specific code. Depending on how the game engine is made up, it may be possible to extend it and add the desired extra functionality.

Lastly I would like to discuss the use of game editors. It is quite impractical to create an entire world just by coding. Nowadays many games make use of specially built editors. These editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the video game when it is released. This gives players the opportunity to build their own worlds to play in and thus can extend the lifetime of the video game. A step further is a so-called mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine. A very well known mod is Counterstrike, which was made based on the Half-Life engine. It even got released as a new (separate) video game.

4.4. Summary of making a 3D video game

When making a videogame, a layered approach is followed (figure 6). Bottom up the layers are API, game engine and game specific code.

An API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for. For game consoles each has its own specialized API(s), since the hardware does not change. For PCs the APIs need to be more general, since there is a lot of difference between the hardware of PCs. Currently the two major PC APIs used are OpenGL and DirectX.

The game engine is built on top of an API and consists of the non-game specific technology. The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. There is no maximum to what an engine should contain. If you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.

What the renderer does is visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed. It is generally the first thing you tend to build when constructing an engine.

The game specific code consists of the game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code. Other features that are usually put in the game specific code are multiplayer/networking options and HUDs/menu's.

When using an existing engine it may be possible to extend it to add extra functionality you need for your game.

Game editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the videogame when it is released. This gives players the opportunity to build their own worlds to play in. A step further is a so-called mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine.

5.Conclusion

In the introduction I asked the following questions:

·What does it take to make a 3D video game?

·What are basic concepts for 3D graphics?

·What are basic concepts for a video game?

·How do you put together the knowledge gained from the previous questions to make a 3D video game?

I will now give answers to each sub question.

·What are basic concepts for 3D graphics?

Firstly, a 3D coordinate system needs to be defined. Secondly, you need to choose how to represent you objects in your coordinate system. You can choose from different kinds of object models with their trade-offs. Currently the polygon (mesh) model is the de facto standard, but this is changing as faster CPU's and more memory become available. Lastly, you need a way to manipulate the objects in your coordinate system. This can be done with affine transformations. The three most basic and commonly used affine transformations for computer graphics are translation, rotation and scaling. Any other affine transformation can be created by applying a properly chosen sequence of the aforementioned basic transformations.

·What are basic concepts for a video game?

The first thing you need to understand is that a video game is just a program with a main function. The main function consists of five states. The three most important states for a video game are connected to each other in the game loop, which is a series of procedures for getting input and displaying output to the player and updating the game. Some interesting aspects from the game loop that are essential for a modern video game are gameplay and artificial intelligence of a video game. However, it all starts with the scenario.

The scenario should describe what is possible in a video game. It should at least describe the interactive elements of the video games, which is the base for the gameplay. Furthermore the scenario can have a plot that connects the interactive elements to each other.

Gameplay determines how a game is played. Gameplay is influenced by a couple of factors: scenario and genre, control and response, viewpoint (camera) and the learning curve and ease of play. Because gameplay is the core of each game, it will determine if the player wants to play it or not.

Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In video games AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game. This

should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. There are different kinds of AI techniques available that can be used for video games.

·How do you put together the knowledge gained from the previous questions to make a 3D video game?

To make a 3D videogame, a layered approach is followed (figure 6). Bottom up the layers are API, game engine and game specific code.

An API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for.

The game engine is built on top of an API and consists of the non-game specific technology. The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. There is no maximum to what an engine should contain. If you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.

What the renderer does is visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed.

The game specific code consists of the game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code. Game editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the videogame when it is released. This gives players the opportunity to build their own worlds to play in. A step further is a socalled mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine.

And now for the answer to the main question:

·What does it take to make a 3D video game?

The layered structure of a video game tells how it is built up and thus what you need to know to make one. You will need to start out by getting (read: buying or building) a game engine. Furthermore, you will always need knowledge of 3D graphics, since in a game engine (renderer) you will always need to deal with it. Finally you must have knowledge of how a video game works. Otherwise you cannot develop the scenario, gameplay, AI et cetera.

The above answer is of course not everything there is to making a video game. Building video games also takes a lot of experience. But when you look at making a game from a theoretical viewpoint as I did, I think that what I have described in this paper makes sense.