

Q3MAP EXPLANATION

20/01/2001

© William 'SmallPileofGibs' Joseph

Thanks to MrElusive and Maj

Deutsche Übersetzung und Anmerkungen von Cassian © 2004

Dieser Text setzt grundsätzliche Kenntnisse über 3dimensionale Geometrie und den Quake3-Editor Q3Radiant oder den GtkRadiant voraus. Ein hervorragende Einführung ins Mapping mit dem Radianten bietet Dangerzones Tutorial!

Definitionen

Geometrie

Grundbegriffe

In einer Map wird ein **Punkt** A als ein Satz Koordinaten (X,Y,Z) gespeichert. Jede Koordinate besteht aus einer Integer-Zahl, also einer ganzen Zahl.

Eine **Linie** AB wird durch zwei Punkte A und B beschrieben. Die Linie verläuft mit unendlicher Länge durch beide Punkte.

Eine **Plane** (Ebene) ABC wird durch drei Punkte A, B und C beschrieben. Eine Plane hat ebenfalls unendliche Ausdehnung und verläuft durch alle drei Punkte. Sie hat zwei Seiten: vorne und hinten. Bei einer senkrecht auf den Systemachsen (X-, Y- und Z-Achse) stehenden Plane handelt es sich um eine **Axial Plane**. Eine Plane steht zum Beispiel senkrecht auf der X-Achse, wenn alle drei Points den gleichen X-Wert haben.

Ein **Area** ist ein endlicher Teil einer Plane, die von Linien begrenzt wird.

Ein **Polygon** ist ein spezielles Area, das von drei oder mehr Linien begrenzt wird.

Ein **Volume** ist ein endlicher Teil eines 3-dimensionalen Raumes. Es wird begrenzt durch vier oder mehr Planes.

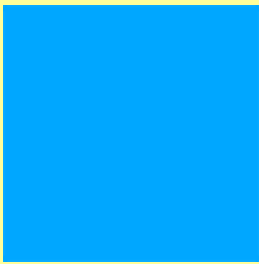
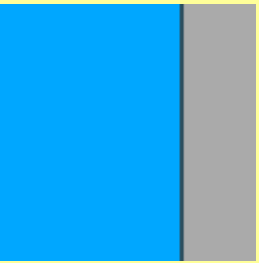

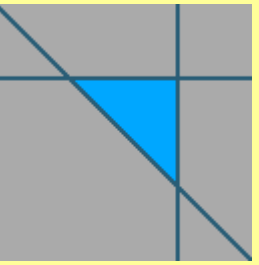
Ein **Vertex** ist ein Punkt (plural: Vertices). Die Koordinaten des Punktes werden allerdings als Floating Point (Fließkommazahl) gespeichert.

Ein **Triangle** (plural: 'Triangles' oder 'Tris') ist ein Polygon mit drei Vertices als Eckpunkten.

Ein **Strip** ist eine streifenförmige Zusammenfassung von mehreren Triangles. Jedes neue Triangle teilt sich Vertices mit dem benachbarten.

Ein **Fan** ist eine sternförmige Zusammenfassung von mehreren Triangles. Jedes neue Triangle teilt sich Vertices mit dem benachbarten.

2dimensional: Areas

	Wir beginnen mit einer Plane.
	Diese Plane können wir mit einer einzigen Line zerteilen.
	Indem wir mehrere nicht-parallele Lines hinzufügen, können wir ein Area auf der Plane definieren.
	Ein Area, das von drei oder mehr Lines auf diese Weise begrenzt wird, ist immer ein konvexes Polygon. Da die Linien unendlich lang sind, ist es unmöglich, so ein nicht-konvexes Polygon zu erstellen.

3dimensional: Volumes

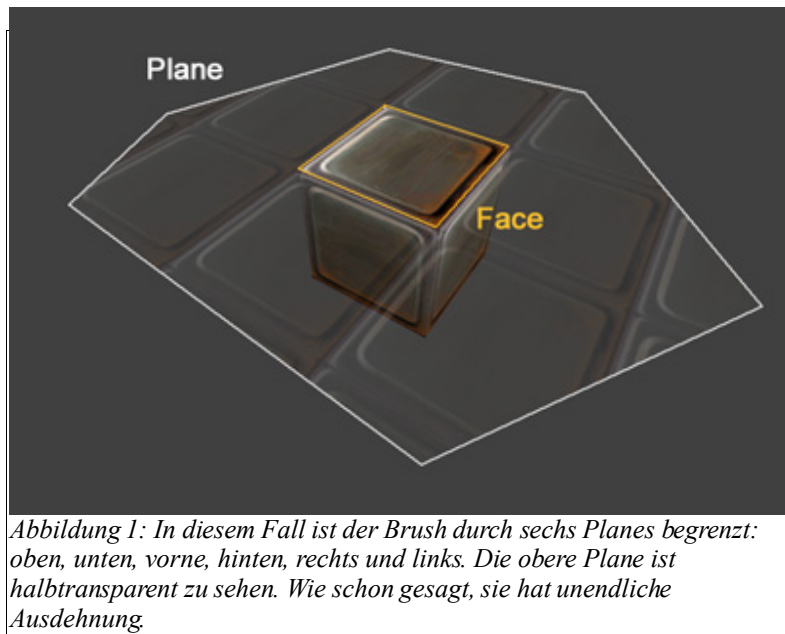
Nehmen wir einen unendlich großen Raum (ein 'Volume') und stellen uns eine Plane vor, die diesen Raum in zwei Teile teilt, vorne und hinten. Diese Teile sind die child-volumes. Man benötigt in diesem Falle vier nicht-parallele Planes um ein endliches Volume zu erstellen. Auch hier ist das endliche Volume immer konvex, also nach aussen gewölbt.

Also: Im zweidimensionalen wird eine Figur durch mindestens drei unendlich lange Linien definiert. Diese Figuren müssen immer konvex sein, also Innenwinkel kleiner als 180° . Dieses Prinzip gilt auch im dreidimensionalen: Hier sind die Begrenzer entsprechend die 'Planes', die auch eine unendliche Ausdehnung haben. Man braucht mindestens vier Planes, um eine Figur im dreidimensionalen Raum zu definieren. Auch diese Figuren sind immer konvex.

Map

Brushes

Ein **Brush** ist ein konvexes Volume. Wie wir eben gesehen haben, ist dieses Volume immer begrenzt von vier oder mehr Planes, wobei jede Plane ein **Face** (Seite) dieses Brushes darstellt.



Jedes Brush-Face hat einen Satz von Eigenschaften: Textur, Oberfläche und Content. **Textureigenschaften** sind: ein Shader, XYZ-Skalierungs- und Stauchungsinformationen sowie Rotation der Textur. Diese Informationen werden in der .MAP-File mit jedem Face gespeichert. Ein Brush-Face kann immer nur einen Satz von Textureigenschaften haben

Meiner Meinung nach werden diese Informationen strenggenommen mit den Planes gespeichert, die das Face definieren. Kleine Spitzfindigkeit von mir...

Hier ein Beispiel, wie ein Brush in der .MAP-Datei dargestellt wird:

```
// brush 6
( 152 -56 32 ) ( 152 -112 32 ) ( 152 -112 24 ) base_wall/blue 0 192 0 -0.218750 0.125000 0 0 0
( 208 -56 32 ) ( 152 -56 32 ) ( 152 -56 24 ) base_wall/bluemeta 0 0 0 0.500000 0.500000 0 0 0
( 208 -112 32 ) ( 208 -56 32 ) ( 208 -56 24 ) base_wall/bluemeta 0 0 0 0.500000 0.500000 0 0 0
( 152 -112 32 ) ( 208 -112 32 ) ( 208 -112 24 ) base_wall/blue 24 80 0 0.300000 0.400000 0 0 0
( 152 -96 88 ) ( 152 -40 88 ) ( 208 -40 88 ) base_wall/basewall 0 0 45 0.100000 0.100000 0 0 0
( 208 -56 56 ) ( 152 -56 56 ) ( 152 -112 56 ) base_wall/bluemeta 0 0 0 0.500000 0.500000 0 0 0
```

Jede

Zeile entspricht einer Plane bzw. einem Face des Brushes. Die drei Klammern am Anfang enthalten die XYZ-Koordinaten jeweils eines Punktes auf der Plane. Damit wird also die Lage jeder Plane definiert. Es folgt der Pfad und Name des Shaders, bzw. der Textur. Die Zahl danach ist die horizontale Verschiebung der Textur gegenüber dem Worldgrid, danach die vertikale Verschiebung. Es folgt Rotation in Grad. Die beiden FloatingPoint-Zahlen danach sind die horizontale und vertikale Stauchung der Textur. Die folgenden Ziffern bezeichnen weitere Eigenschaften, z.B. dass der Brush 'detail' ist.

Oberflächen- und Contenteigenschaften werden in einem externen Shader-Script gespeichert. Andererseits werden wiederum manche Contenteigenschaften trotzdem in der .MAP-File gespeichert, zum Beispiel, dass der Brush 'detail' ist.

Bezier Patch Oberflächen

Bezier Patches sind eine bestimmte Art der parametrischen Oberflächen. Parametrische Oberflächen sind Oberflächen, die durch einen Satz von Parametern beschrieben werden. Der Wert dieser Parameter bestimmt das Aussehen dieser Oberfläche. Der Parameter für Bezier Patches in Quake3 ist der LOD, der Level Of Detail.

Patches werden in Quake3 in Triangles gerendert, also müssen die Curvestessalisiert werden, dh. sie werden in eine bestimmte Anzahl von Triangles 'unterteilt'.

Je höher der LOD, desto mehr wird die Curve unterteilt und es entstehen mehr Triangles durch die Tessalation. Der LOD kann so eingestellt werden, dass er mit steigendem Abstand des Betrachters abnimmt. Bei Quake3 kontrolliert „r_lodCurveError“, wie stark sich der LOD mit der Entfernung ändert,

„r_subdivisions“ bestimmt den maximalen Wert für den LOD.

Ein Patch wird als zweidimensionale Matrix aus Punkten gespeichert. Eine zweidimensionale Matrix entspricht einer Tabelle, die eine Reihe von Wertsätzen enthält, die in Zeilen und Spalten organisiert sind. Jeder Punkt wird mit zwei Wertsätzen beschrieben. Ein Satz enthält die XYZ-Koordinaten dieses Punktes im dreidimensionalen Raum der Map. Der andere Satz enthält die UV-Koordinaten, die die Position des Punktes auf dem entsprechenden Shader beschreibt. Genauso wie ein Brush kann ein Patch nur einen Shader haben.

Shader

Ein **Shader** ist eine die Zusammenfassung von Einstellungen für eine Oberfläche. Er kontrolliert, wie die Oberfläche von den Compilern behandelt und von der Quake3-Engine gerendert werden soll. Jede Oberfläche hat einige Grundeinstellungen, die benutzt werden, solange noch kein externer Shader spezifiziert ist.

Die Kombination einer gewissen Anzahl von Voreinstellungen kontrolliert die Behandlung durch den Compiler. Nähere Informationen gibts im Shader Manual².

„**SurfaceParm**“ ist das Keyword für die allgemeinen Shadereinstellungen. Diese Shadereinstellungen sind entweder Content- oder Oberflächeneigenschaften des Brushes

Wenn ein Shader mit einer **Contenteigenschaft** auf das erste Face (Face 0) angewendet wird, gilt diese Eigenschaft für den gesamten Brush. Um unvorhersehbare Ergebnisse zu vermeiden ist es sinnvoll, diese Eigenschaft auf alle Faces dieses Brushes zu übertragen.

Eine **Oberflächeneigenschaft** gilt nur für die eigentliche Oberfläche, bzw. das eigentliche Face.

Wichtige SurfaceParms:

Content

solid	DEFAULT, Brush ist solid, wirft Schatten, blockt Player und alle Entities
nonsolid	Brush ist durchlässig
structural	DEFAULT, Brushfaces teilen den BSP-Baum, blockieren die Sicht aber nur, wenn sie keine trans-Eigenschaft hat.
detail	Brushfaces teilen weder den BSP-Baum, noch blockieren sie die Sicht
playerclip	blockt den Player
trans	Brush wirft keine Schatten.

Surface

nodraw	Das Face wird beim kompilieren entfernt.
nolightmap	Für diese Oberfläche wird keine Lightmap berechnet.

Entities

Eine Quake3-Map besteht aus verschiedenen **Entities**, nummeriert von 0 aufwärts. Eine Entities dürfen Brushes und BezierSurfaces beinhalten. Defaultmäßig gehören alle Brushes und BezierPatchSurfaces zum sogenannten „**Worldspawn**“-Entity (entity 0). Der Worldspawn ist die solide Hülle der Map. Er enthält alle anderen Entities. Alle Entities haben einen **Origin Point** und ein achsenparalleles Begrenzungsvolumen (aka **bounding box**). Der Origin des Worldspawns ist der Punkt (0,0,0).

Alle Entities haben verschiedene Eigenschaften, die als Key/Value-Paar abgespeichert werden. Die Werte können sowohl Zahlen als auch Strings sein.

2 http://www.qeradiant.com/manual/Q3AShader_Manual/index.htm

Mapobjects

Jedes Mapobject ist eine Sammlung aus Triangle-Oberflächen, die in Strips organisiert sind. Die Strips werden als Frame in einer einzigen MD3-Datei gespeichert. Mit einer BezierPatch-Oberfläche haben Mapobject-Strips gemeinsam, dass sie nicht structural und nur einem Shader zugeordnet sein können. Neben dem Namen dieses Shaders wird ein Satz mit den Texturkoordinaten in der .MD3-Datei gespeichert. Es handelt sich um das gleiche Format, wie das für die Items und die Playermodels. Deswegen erfordern die Models auch kein Pre-Processing, außer dass sie in der ersten Compiler-Stage zu der BSP hinzugefügt werden

Mapobject-Strips können weder Content-Eigenschaften noch eine Lightmap haben, aber sie können abhängig von ihrem Shader Licht emittieren oder Schatten werfen.

Der Kompilierungsprozess

Der Kompilierungsprozess hat vier Stages: **BSP**, **VIS**, **LIGHT** und **BSPC**

Stage 1: BSP

Kompilierte Q3-Levels werden als .BSP-Datei gespeichert. Der wichtige Teil dieses Prozesses ist die Erstellung des BSP-Baums aus den Brushes der Map.

Was macht'n die BSP?

Der durch den Player betretbare Raum innerhalb der Map wird in konvexe, durch Planes begrenzte Volumens aufgeteilt. Diese Volumens werden **Leaf Nodes** genannt. Diese Nodes werden in einem Binärbaum gespeichert, dem sogenannten **BSP-Baum**³.

Uffjemerkt! Der durch den Player betretbare Raum in der Map ist überall dort in der Mapwelt, wo es keinen Brush gibt, der sowohl solid als auch structural ist. Es ist der Raum, den ein Player mit 'noclip' erreichen kann, ohne einen 'structural'- oder 'solid'-Brush überwinden zu müssen. Alle Brushes in der Map sind erstmal solid und structural, solange sie keine anderen Contenteigenschaften verpasst bekommen haben. Non-'structural' Eigenschaften sind 'detail', 'playerclip' und 'trans'. Non-'solid'-Eigenschaften sind 'Water', 'Lava' und (wen wunderts?) 'nonsolid'.

'Detail'-brushes werden durch 'surfaceparm detail' im Shader erzeugt, oder indem man sie direkt im Radiant 'detail' macht. ('detail'-Brushes können mit STRG + D unsichtbar/sichtbar geschaltet werden.) Einige der Common-Texturen sind zwar structural, aber nicht visible. Dazu gehören: CAULK, HINT, nodraw, nonsolid und AREAPORTAL. Der Rest ist entweder non-structural, wird nur auf Entities oder normalerweise garnicht gebraucht.

Wie und warum wird die BSP erzeugt?

Eine **Map** ist ein dreidimensionales Raumvolumen über +/- 4096 Units ab dem Origin in allen drei Koordinatenachsen. Dieses Volumen enthält weitere kleinere, konvexe und solide Volumens, die durch Planes begrenzt werden (in diesem Fall structural Brushes).

Ziel ist, möglichst wenig Splits zu benutzen, um die Map in **konvexe Volumens** aufzuspalten. Diese Volumens dürfen keine structural Brushes (bzw. Planes) enthalten. Ein einfacher, würfelförmiger Raum ist ein konvexes, von sechs Planes begrenztes Volumen. Für die VIS-Berechnung sind konvexe Volumens sehr wichtig, weil man einfach und schnell feststellen kann, ob sich zwei konvexe Volumens gegenseitig sehen.

Die konvexen Volumens werden als LeafNodes in einem Binärbaum gespeichert. So ist relativ einfach festzustellen, in welchem LeafNode die Viewpoints oder Objekte (z.B. Entities, Tris, Patches) sich befinden. Dieser Baum wird *Binary Space Partitioning Tree*, oder auch **BSP-Tree** genannt.

Der BSP-Tree wird durch eine rekursive Operation erzeugt. Eine rekursive Operation ist eine Operation, die sich selber wieder aufruft. Diese spezielle Operation wird auf ein Volumen angewendet, wobei dieses Volumen

³ http://www.pohlig.de/Unterricht/Inf2002/Tag43/28.11_Was_Versteht_Man_Unter_Einem_Binaerbaum.htm

am Anfang die gesamte Map ist:

```
if (Volume enthält structural Brushes) {
    zerteile das Volume entlang einer 'structural'-Plane
    beginne die Funktion erneut mit dem ersten Child-Volume
}else {
    erzeuge LeafNode aus diesem Volume
    gehe den Baum eine Stufe hoch
    if (weiteres Child-Volume vorhanden) {
        wiederhole Funktion mit diesem Volume
    }
}
Beende Funktion
```

Aufgepasst:

- Die Split-Plane zerteilt das Volume nicht immer genau in die Hälfte
- Axiale (parallel zu den Koordinatenachsen) Planes werden den nicht-axialen Split-Planes bevorzugt.
- Es wird die Split-Plane ausgewählt, die durch die höchste Anzahl von Brushs schneidet.

Andererseits ist es aber wichtig, das jedes Volume ungefähr in der Hälfte geteilt wird, damit jedes LeafNode ungefähr den gleichen Abstand zum Ursprung des BSP-Trees hat, er also möglichst gleichmäßig aufgebaut ist.

Wie kann ich die Aufteilung steuern?

Alle 'structural'-Brushface-Planes, auch vollkommen unsichtbare oder mit 'nodraw'-Shader belegten faces, können den BSP-Tree teilen und neue LeafNodes erzeugen. LeafNodes werden niemals zusammengefasst, sogar wenn das Ergebnis ebenfalls ein konvexes LeafNode ergeben würde. Ideal wäre es, wenn alle Brushfaces alle 128 Units vorkommen würden und diese senkrecht zuden Achsen wären. Jeder komplexere Brush kann 'detail' gemacht werden, alle versteckten/unsichtbaren Brushfaces werden durch Belegung mit common/caulk 'nodraw'. Den gleichen Effekt erzielt man mit BezierPatch-Oberflächen, die ja weder vis blocken noch andersichtbaren Oberflächen als die vordere haben.

Hint-Brushes sind structural, trans und nonsolid. Das heißt: obwohl sie in einem Level vollkommen unsichtbar sind, zerteilen sie wie andere 'structural'-Brushfaces den BSP-Tree beim compilen. Ein sehr großer Hint-Brush wäre eine idealer Kandidat, den BSP-Tree zu zerteilen. Durch hinzufügen von Hint-Brushs mit axialen Faces in Verlängerung von anderen 'structural'-Brushfaces würde die Anzahl von zusätzlichen SplitPlanes und LeafNodes verkleinert.

Wenn man eine Map nach dem **128-Unit Grid** ausrichtet, kann man den Visibilty-Prozess sehr vereinfachen. Wenn die LeafNodes alle 128-Unit-Würfel sind, dann ist es Für den Designer sehr einfach vorauszusagen, ob ein Teil der Map von einem anderen sichtbar ist. In diesem Fall ist es dann sehr einfach, die Sichtbarkeit in der Map einzuschränken, indem man einfache axiale Hint-Brushs reinbaut, ohne einen merklichen Anstieg der VIS-Kompilierungszeit hinnehmen zu müssen. Auf diesem Skelett aus 'Structural/NoDraw'-Brushes kannman dann die komplexere Architektur aufbringen, indemman 'detail'-Brushes und BezierPatch-Oberflächen verwendet, die ja die Figur der Nodes nicht beeinflussen.

Was ist ein Leak?

Aus Compiler-Sicht ist eine Quake3-Map ein hohler Raum, bestehend aus mehreren konvexen Volumes (LeafNodes) in einer festen, äusseren Hülle. Die LeafNodes, die sich ausserhalb dieser Hülle befinden, werden entfernt, nachdem der BSP-Tree erstellt wurde.

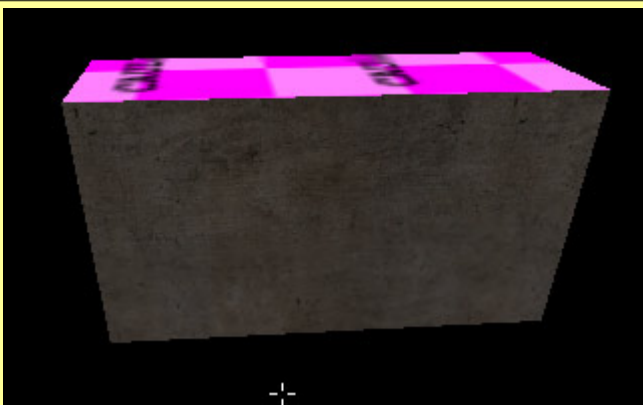
Wenn man jedoch die Entities innerhalb der Map nicht vom 'void' ausserhalb des Radiant-Grids trennen kann (durch diese Hülle) spricht man von einem 'Leak'. Wenn ein Leak auftritt, dann wird eine detaillierte Fehlermeldung von q3Map ausgegeben, wenn man -v als Option benutzt. Außerdem speichert es den Pfad des Leaks in einer .LIN-datei. Normalerweise wird das Leak im Radiant als roter Pfad angezeigt, der vom Origin eines Entities zu einem LeafNode führt, das den Bereich außerhalb des Radiant-Grids berührt, ohne einen 'solid'-Brush zu durchkreuzen.

Was ist eine .PRT-Datei?

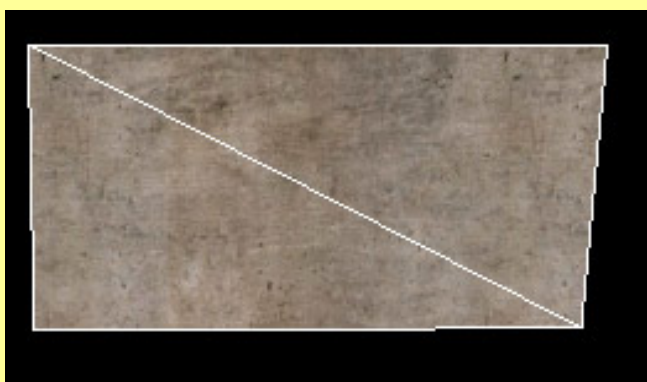
Die LeafNodes, die nach dem Entfernen der äußeren LeafNodes im BSP-Tree verbleiben, werden entweder von anderen LeafNodes begrenzt, oder sie berühren einen 'structural'-Brush. An den Stellen, wo ein LeafNode ein anderes berührt wird ein Portal erstellt. Man kann sich ein **Portal** als Fenster zwischen zwei LeafNodes vorstellen. Die Informationen über diese Portals werden in der BSP-Stage nicht benötigt, sie sind aber für die VIS-Berechnung um so wichtiger. Deswegen werden sie in einer PRT-Datei gespeichert, der **Portal File**.

Wie verwandeln sich Brushes und Patches in Triangles?

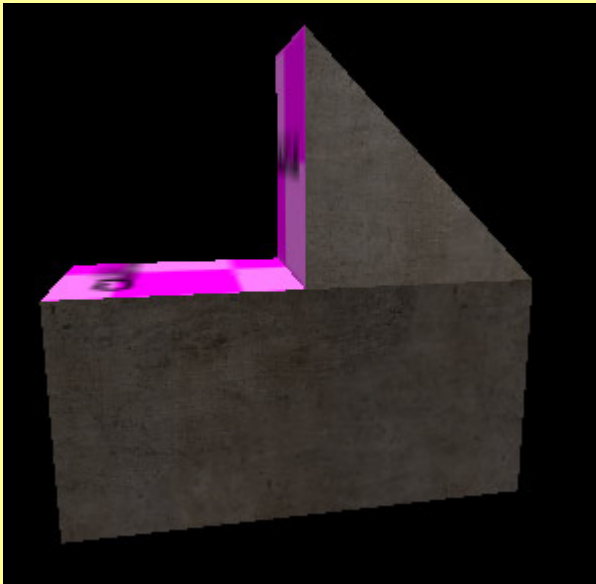
Bei Brushes wird für jedes sichtbare Brushface, das ein oder mehrere konvexe LeafNodes berührt, ein konvexes Polygon erstellt. Brush Faces mit 'surfaceparm nodraw' werden ignoriert. Dort wo die Seite eines Polygons mit einem Vertex (also dem Eckpunkt) eines weiteren Polygons zusammentrifft und damit eine **T-Junction** erzeugt, wird ein neuer Vertexpunkt erstellt. Diese neuen Vertexes reparieren die T-Junction und erzeugen einen Punkt, wo sich drei Polygonkanten treffen.



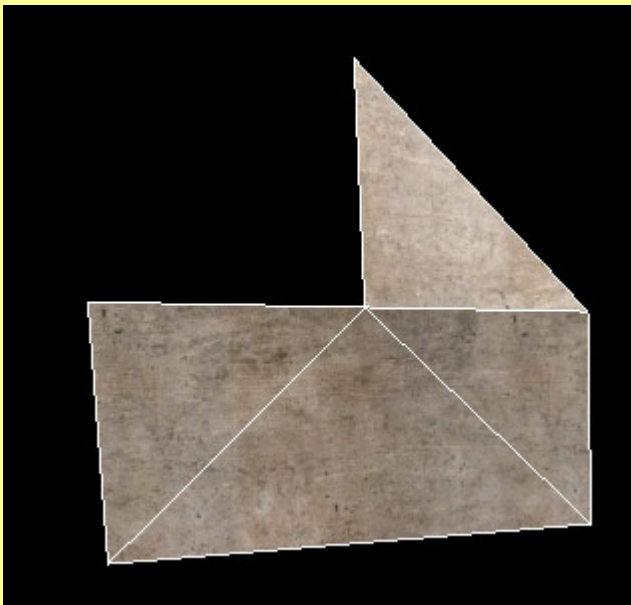
Nehmen wir als Beispiel diesen Brush. Ein Face des Brushes ist mit einer einfachen Textur belegt. Die fünf restliche Faces sind belegt mit 'common/caulk'.



Das Face hat vier Vertices, jeweils eins an jeder Ecke. Dieses Face wird bei der **Tessalation in Triangles** zerlegt. Dabei wird bei einem Viereck das Polygon entlang der Diagonalen zerteilt. Alle anderen Schnittpunkte entlang der Vertices würde keinen Sinn machen, eine Zerteilung entlang benachbarter Vertices **keine** Zerteilung wäre. Hat man die kompilierte Map mit dem Befehl /devmap mapname gestartet, so kann man sich die entstandenen Tris mit dem Befehl /r_showtris 1 anzeigen lassen:



Wie man sehen kann sind aus dem viereckigen Face zwei Tris entstanden.
Setzen wir einmal ein Dreieck auf diesen Brush.
Dieses dreieckige Face muß nicht weiter tessalisiert werden, das kann es auch garnicht, denn hier gibt es **nur** benachbarte Vertices.



Man könnte jetzt meinen, dass die neue Form aus drei Triangles besteht. Das ist falsch! Es sind **vier!**

Der linke untere Vertexpunkt des Dreiecks erzeugt eine T-Junction: Die obere Kante des Vierecks verläuft durch den Vertexpunkt. Um diese T-Junction zu beseitigen, wird am Face des viereckigen Brushes ein **zusätzlicher** Vertexpunkt erzeugt. Würde das Face nur wie im ersten Beispiel tessalisiert, hätte das obere Tris des viereckigen Faces **vier** Vertexpunkte, was nicht erlaubt ist (dann wärs ja auch kein **Triangle** mehr). Deswegen wird der Brush anders tessalisiert und es entstehen drei Tris statt zwei.

Wenn jetzt ein Vertex auf einer Kante liegt, die nicht zerteilt werden kann, entsteht ein kleiner Bruch in der Kante. Diese Brüche nennt man **Sparklies**, da sie so aussehen, wenn man durch diesen Bruch einen hellen Hintergrund sieht, der hindurchschert.

Am Ende wird jedes Polygon in Triangles zerteilt und als ein OpenGL-Primitive gespeichert. Dieses Primitive nennt man 'Triangle-Strip' oder manchmal auch 'Triangle-Fan'

Patch-Oberflächen werden während der BSP-Stage als Sammlung von Punkten behandelt und erst dann in Triangle-Strips verwandelt, wenn die Map geladen wird. Deswegen können T-Junctions, an denen Patch-Oberflächen beteiligt sind, nicht automatisch repariert werden.

Für jeden Vertexpunkt werden **Texturkoordinaten** erstellt. Diese Koordinaten sind zwei Wert U und V (UV-Koordinaten oder 'UVs'), die die Position dieses Vertexpunktes auf dem Texturbild als Prozentwerte festlegen. Diese Werte sind dezimal normalisiert (FloatingPoint), d.h. ihr Wert liegt zwischen 0 und 1: Wenn eine Textur 256*128 Pixel groß ist und ein Vertex hat die UV-Koordinaten (0.5, 0.75), dann wird das

Texturpixel (**Texel**), das bei 50% der Breite 256 und 75% der Höhe 128 liegt, auf dem Vertex gezeichnet. Guck! Jedes Brushface und jede Patch-Oberfläche ist ein einzelnes Primitive. Deswegen können sie auch ihren eigenen Satz Texturkoordinaten und Shader-Einstellungen haben.

Stage 2: VIS

VIS ist die Abkürzung für **Visibility**. Der relevante Teil der VIS-Berechnung ist die Erstellung der sogenannten **PVS-Tabelle** für die Portals in der Map. Wurde die Map mit FULL-VIS kompiliert, dann erscheinen verschiedene Areas der Map oder verschwinden wieder, während sich der Standpunkt des Spielers in der Map bewegt. Welche Areas der Map sichtbar sind, hängt vom Standpunkt des Spielers ab.

Wie wird das PVS erstellt?

Jedes Portal wird gegenüber jedem anderen Portal auf Sichtbarkeit überprüft. Portal 2 ist von Portal 1 aus sichtbar, wenn man eine direkte Linie zwischen jedem Teil der beiden Portale ziehen kann, ohne einen 'structural'-Brush zu kreuzen. Jedes Portal erhält also eine Liste von Portals, die von ihm aus sichtbar sind. Diese Liste ist das **Potential Visible Set (PVS)** und wird in der PVS-Tabelle gespeichert.

Was macht -vis -fast?

AKA 'bsp_fastvis'. Wenn man mit -vis -fast kompiliert, so wird keine PVS-Tabelle erstellt. Damit bleibt sozusagen jedes Portal von allen aus sichtbar.

Welchen Einfluß haben Portals auf die VISIBILITY?

Jedes LeafNode hat ein oder mehr Portals (so lange nicht nur ein Node im BSP-Tree gibt). Kann ein Portal, das zum Node 1 gehört, ein Portal sehen, das zur Node 2 gehört, dann ist Node 1 von Node 2 sichtbar. Wenn nun der Standpunkt des Players in Node 1 liegt, dann wird auch jedes Objekt in Node 2 gezeichnet. Befindet sich der Player in LeafNode X, dann wird jedes Objekt gezeichnet, das sich in den LeafNodes befindet, die von LeafNode X sichtbar sind.

Objekte sind in diesem Fall BrushFaces, BezierPatches oder Entities. Ein Objekt kann sich in mehreren LeafNodes gleichzeitig befinden. Ein BrushFace wird gezeichnet, wenn sich irgendein Teil in einem sichtbaren LeafNode befindet. Ein Bezierpatch wird gezeichnet, wenn sich irgendeiner seiner Kontrollpunkte in einem sichtbaren LeafNode befindet. Ein Entity wird gezeichnet, wenn sich irgendein Teil seiner BoundingBox ein sichtbares LeafNode berührt.

Um mal zusammenzufassen: Solide 'structural'-Brushes blocken die VISIBILITY zwischen den LeafNodes. Curves, 'detail'-Brushes und Entities tun das nicht. Die Visibility zwischen zwei Bereichen wird nur geblockt, wenn sie vollkommen voneinander abgeschottet sind. Im Allgemeinen werden desto mehr Objekte gezeichnet, je mehr LeafNodes von dem Standpunkt sichtbar sind, an dem der Player steht. Das Ziel bei der Optimierung der Visibility muß deswegen sein, dass möglichst wenig LeafNodes von einem LeafNode sichtbar sind.

Wenn der BSP-Tree nur eine kleine Anzahl großer LeafNodes hat, oder wenn die LeafNodes ineffizient angeordnet sind, dann wird auch mehr von der Map gezeichnet. Man kann die Anzahl der LeafNodes reduzieren, indem die Menge der einzelnen 'structural'-Planes in der Map verringert wird. Dies erreicht man, indem man diejenigen Planes 'detail' macht, bei denen die 'structural'-Eigenschaft unnötig ist.

Man kann die visibilität kontrollieren, indem man 'Hint'-Brushes benutzt und dadurch genau festlegt, wo die Portals liegen und kleinere LeafNodes nur da erzeugt, wo sie gebraucht werden.

Wie kann ich VIS effizienter machen (und damit die VIS-Zeit verkürzen)?

Die VIS-Kompilierungszeit ist ungefähr proportional zu Anzahl der Portals. Diese Anzahl wird beim Starten der VIS-Kompilierung als "**numportals xxxx**" angezeigt. "**visdatasize**" ist die Größe der PVS-Tabelle. Sie ist auf ungefähr 2MB limitiert.

Viele LeafNodes bedeuten viele Portals und das bedeutet wiederum eine lange VIS-Zeit. Die Anzahl der

LeafNodes hängt ausschließlich von der Komplexität des BSP-Trees ab.

Die Lösung: **'Detail'-Brushes**. Ein 'detail'-Brush hat keinen Einfluss mehr auf den BSP-Tree. Damit wird die Anzahl der erstellten LeafNodes reduziert.

Um einen Brush 'detail' zu machen, muß man ihn selektieren und strg + m drücken oder durch Auswahl von Selection > Make detail. Man kann die 'detail'-Brushes mit strg + d ein- und ausblenden oder über das View > Show menu. (Im GtkRadiant ist dies das 'Filter'-Menü)

Der Nachteil an den 'detail'-Brushes ist, dass eine zu starke Vereinfachung der LeafNodes die visibility-Effizienz beeinträchtigt (siehe VIS-Zusammenfassung)

Neue Lösung: **'Hint'-Brushes** – Ein Hint-Brush (common/hint) ist zwar im Spiel unsichtbar, aber er ist trotzdem 'structural', also wird er auch den BSP-Tree beeinflussen und die Erstellung von LeafNodes auslösen. Das gibt eine starke Kontrolle darüber, wo die LeafNodes erstellt werden. Wird ein Hint-Brush in einem freien Bereich erstellt, wird die Erstellung eines LeafNodes an der Stelle des Hint-Brushes ausgelöst. Hint-Brushes dürfen andere 'structural'-Brushes oder Hints überlappen, um mehr LeafNodes zu erstellen oder eine bestimmte Gruppe von LeafNodes zu isolieren. Der Shader der Hintbrushes (common/hint) hat außerdem die Eigenschaften 'nonsolid' und 'nodraw'.

Hints sind am vorteilhaftesten, wenn sie Planes axial schneiden und diese Schnitte in einer Linie mit anderen 'structural'-Brushes sind. Dies vergrößert den Bereich, der von den 'structural'-Brushes verdeckt wird und verringert die Anzahl und Größe sichtbarer LeafNodes.

Die Kombination von 'detail'- und Hint-Brushes kann die VIS-Zeit und r_speeds in fast jeder Map verringern. Trotzdem muß dies schon von Beginn an beim Aufbau der Map im Hinterkopf behalten werden. Im nachhinein ist das eine Heidenarbeit.

Uffjemerkt: Ein 'detail'-Brush blockt kein VIS mehr, also sollte man die VIS-block Brushes nicht detail machen.

Stage 3: LIGHT

Dies ist die **Lightmap**-Stage, in der die Lightmaps für jede Oberfläche der Map erzeugt werden. Die Lightmap hat keinen Einfluss auf den BSP-Tree, die Hints, VIS oder die r_speeds.

Wie funktionieren Lightmaps?

Der LIGHT-Algorithmus von q3map.exe erstellt ein Lightmap-Pixel für jeweils 16 Gameunits auf einem Brush und jeweils 20 Units eines Patches. Diese Pixel werden in 128*128 Pixel großen Seiten gespeichert. Die Farbauflösung der Lightmaps ist 24bit RGB. Die Werte der Lightmap werden mit den RGB-Werten der Texturen multipliziert.

Wie werden Lightmaps erzeugt?

Alle Lightmap-Pixel sind zu Anfang schwarz (RGB 0 0 0). Vom Zentrum jedes Lightmap-Pixels (entspricht der Position der zugehörigen Brush-Plane in der Map) wird eine direkte Linie zur Position jeder punktförmigen Lichtquelle gezogen. Der Abstand zwischen dem Lightmap-Pixel und der Lichtquelle bestimmt die Helligkeit, die dieses Licht dem Pixel hinzufügt. Wird die Linie durch irgendeinen sichtbaren, nicht-transparenten Brush unterbrochen, hat diese Lichtquelle keinen Einfluss auf dieses Lightmap-Pixel. Die Zeit, die die LIGHT-Stage benötigt, ist proportional zur Anzahl der Lightmap-Pixel multipliziert mit der Anzahl an punktförmigen Lichtquellen.

Was ist denn dann mit den SurfaceLights?

SurfaceLights werden in einzelne PointLights unterteilt. Die Größe der Unterteilung richtet sich nach dem q3map_lightsubdivide-Wert. Sein Defaultwert ist 64, in diesem Fall wird alle 64 Units ein PointLight auf der Textur erstellt. Sie haben die gleiche Auswirkung auf die Lightmap wie normal PointLights.

Was ist der Effekt von -light -extra

Bei der Option -extra wird zusätzlich zur normalen Lightmap-Berechnung noch jeweils eine Linie von vier verschiedenen extra-Pixeln zuden Lichtquellen gezogen. Der Durchschnittswert dieser Werte ist dann der neue Lightmap-Pixel-Wert. Dies glättet zwar gezackte Schatten, macht aber die Dauer der Lightmap-Stage 5mal so lang.

Stage 4: BSPC

Bei dieser Stage wird das BSPC-Tool benutzt (bsp.exe -bsp2aas mapname.bsp), das eine **Area-File** (.AAS-Datei) erstellt. Diese Area-File wird von Bots zur Navigation in der Map benutzt.

BSPC ist im Gegensatz zu den anderen Compile-Stages ein eigenständiger Prozess, und hat weder einen Effekt auf diese Stages, noch benutzt er die Informationen aus diesen Stages. BSPC benötigt lediglich eine .bsp-Datei, in der sich kein Leak befindet.

Wie liest BSPC die Map?

BSPC untersucht die .bsp-Datei, um eine Liste der Faces zu machen, die Bots blockieren. BezierPatches werden in grade Flächen umgewandelt, die sogenannten **CurveBrushes**, die dann genau wie normale BrushFaces behandelt werden. Contenteigenschaften, die Bots blockieren sind z.B.: 'structural', 'detail', 'trans' oder 'playerclip'. Von allen Faces, die einen Player blockieren, wird angenommen, dass sie auch die Bots blockieren sollen, also werden sie genauso als 'sdid' behandelt und alle weiteren Contenteigenschaften werden vergessen.

Was hat es mit der "Brush CSG"-Phase der BSPC auf sich?

Während dieser Phase werden unnötige Brushes verworfen. Brushes, die sich **innerhalb** von anderen Brushes befinden, werden durch **CSG-Subtracting** des größeren vom kleineren entfernt. Je mehr Brushes BSPC ignorieren kann, desto besser – hier kann man große, einfach geformte ClipBrushs verwenden, um jegliche komplexe Architektur aus mehreren Brushes für das BSPC zu vereinfachen.

Wie wird die Area-File erstellt?

BSPC zerteilt den von Bots betretbaren Raum der Map in konvexe Volumes, die sogenannten **Areas**. Dieser Prozess gleicht dem BSP-Prozess. Jedes Area muß konvex sein, genauso wie die LeafNodes im BSP-Tree. BSPC versucht darüber hinaus die Anzahl der Areas zu verkleinern, in dem er Areas miteinander verbindet, die zusammen ein neues, konvexes Area ergeben würden. Die Areas werden von BSPC in der .aas-Datei in Gruppen gespeichert, den **Clustern**. Ein Cluster ist eine Gruppe aus verbundenen Areas, die von anderen Clustern durch feste Wände oder **Clusterportalen** getrennt sind. Ohne die Clusterportale wären die Maps einfach ein einziger, großer Cluster aus Areas.

Die Cluster/Area-Informationen stehen nach dem Erstellen der .aas-Datei in der bspc.log.

Achtung: Von Bots betretbarer Raum ist überall innerhalb der Map, wo kein solider PlayerClip-Brush ist. Die Contenteigenschaft "PlayerClip" gehört zB. zu 'solid' oder 'playerclip'.

Wie benutzen die Bots die Area-File

Die Bots benutzen die Area-File, um sich zwischen den Entities zu bewegen. Sie müssen jederzeit die Informationen über jedes Area, das zu dem Cluster gehört, in dem sie sich gerade befinden, berechnen. Sind sehr viele Areas in ihrem Cluster gespeichert, so wird die CPU-Last der Bot-Navigation merklich höher, wenn sie sich in diesem Cluster befinden. Sind mehr als 1000 Areas in diesem Cluster, so drückt dies sehr auf die Performance – unter 500 ist ein wesentlich besserer Wert. Je weniger Areas im größten Cluster, desto besser.

Wie erstelle ich Cluster- Portals?

ClusterPortal-Brushes werden benutzt, um die Erstellung von Clustern durch BSPC zu steuern, indem man der potentielle ClusterPortals "vorschlägt". ClusterPortal-Brushes sollten nicht mit AreaPortals verwechselt werden, da es dort ein paar wichtige Unterschiede in der Verwendung gibt. ClusterPortal-Brushes sollten dünne, etwa 32 Units dicke, axiale Brushes sein. Um als ClusterPortal von BSPC angenommen zu werden, müssen zwei gegenüberliegende Faces jeweils einen anderen Cluster berühren. Der ClusterPortal-Brush muß in einem Türdurchgang oder einer horizontalen Passage platziert werden, wo der Bot frei durchlaufen kann, anstatt durchzufallen. Die beiden gegenüberliegenden Faces sollten die gleiche Form haben, die vier anderen sollten 'solid'- oder ClipBrushes berühren.

Achtung: Genauso wie Areaportals müssen ClusterPortals zwei Clustern komplett voneinander trennen. Wenn ein Bot Cluster 2 von Cluster 1 erreichen kann, ohne durch ein ClusterPortal hindurch zu müssen, dann sind beide Cluster verbunden, und werden zu einem Cluster.