Modeling and Animating for Half-Life

An Overview of the *Half-Life* Modeling Process

This document is an introduction to modeling and animating characters for the *Half-Life* engine. In it we will discuss the basic techniques for creating a model, attaching into a skeleton, animating it, and exporting it into the *Half-Life* engine.

All of the models in the shipping version of *Half-Life* were created with 3-D studio Max from Kinetix Inc., and the export tools included with this SDK currently only support Max. In principle, however, there's no reason why any other 3-D package could not be used with if you can write a simple file exporter. Models could even be generated programmatically by programs such as Mathematica, or even by Java scripts. Valve encourages developers to write translators for other popular 3-D packages and may be able to provide advice to teams seeking to develop their own export plug-ins. For the remainder of this tutorial we will be discussing tools and procedures in the 3dStudio environment; however the basic principles involved will not change if the model and animation data originate in other programs.

Three steps are involved in moving the model data from the 3DStudio into the *Half-Life* engine.

First, a text based intermediate file known as a "Studio Model Data" or SMD file, is exported from the modeling package.

Next, a control script which will direct the final model generation program is written. This is also a text file, known as a QC file because it evolved from the old Quake engine QuakeC scripting system.

Finally the command line program `studioMDL.EXE` analyzes the QC file and converts the .smd file(s) into a *Half-Life* .MDL model file complete with textures, animation data, and game data such as the handles which the AI system uses to call animations.

Understanding the nature of this process is indispensable for easily and quickly making *Half-Life* models. Here is a more detailed look at each of steppingstones from 3dStudio to the *Half-Life* engine:

Exporting from 3Dstudio

There are two paths for exporting data from studio Max to the SMD files. Included with the *Half-Life* SDK is a Max plug-in SMDEXP.DLO. This is available in versions supporting Max 1.2 through 3.1; source code is also included so you can recompile new versions as necessary or make customizations. Installing the plug-in requires only that you drop the appropriate plug-in file(s) into the plug-ins folder of your 3Dstudio Max directory and restart Max . To access the exporter you simply choose "Export…" from Max's File menu. The standard exporter requires the *Physique* plug-in from Kinetix's Character Studio to enable skeletal animation; without *Physique* only unarticulated models without skeletons can be exported.

Also included with the SDK is an SMD exporter based on Kinetix's Max Script scripting language. This is included primarily for users to do not have access to Character studio. The Max Script exporter supports the standard Max r3.x plug-ins *Skin* and *Linked Xform* as alternative methods for attaching for vertices to bones. The Max Script exporter requires Max version 3.0 or higher.

SMD files

SMD files come in two “flavors.” The *Reference SMD* file is a complete snapshot of the model, including its geometry, its skeletal structure, its texture, and the links between its mesh vertices and its skeleton. Reference SMDs do not contain any animation data -- they produce only a static image of the complete model at one point in time.

Animation SMDs, on the other hand, contain only animation data and enough skeletal information for the StudioMdl program to be sure that the animation data matches the skeleton of the reference model. As you can see any animated model will require at least two SMDs – and a glance at the *Half-Life* source files in the SDK will show you that most models include a very large number of animation files. Most models however will use only a single reference SMD.

As an aside it should be noted that since both kinds of SMD files are text-based they can be edited by hand. There may situations where this is a useful time-saver. The SMD file format is included in an appendix to this document.

An Alternative To 3DStudio

MilkShape by Mete Ciragan is a shareware modeling program with excellent support for *Half-Life* modeling and the ability to export SMD files. A version of MilkShape is available is included with the *Half-Life* SDK 2.1 or online at http://www.swissquake.ch/chumbalum-soft/ms3d1x/

There are MilkShape tutorials online at:

http://www.swissquake.ch/chumbalum-soft/ms3d1x/docs/ms3dbasics/ms3dbasics-01.html.

MilkShape is produced independently and Valve takes no responsibility for the program or its performance. Contact and copyright information are included with the program in the SDK.

QC file

It may be helpful to think of the QC files as a kind of table of contents for model projects: the QC provides a complete list of all the assets which will be included in a finished model. The primary function of the QC is to tell the StudioMdl program where to find textures and SMD files. This SDK includes a Max Script which can help you generate simple QC files.

The QC is also the place where game-engine data is specified. For example, the QC file tells the engine which bone controls the scientist model’s mouth, or where the Gargantua’s glowing eye sprite attaches, or how big a target the headcrab is. These topics are dealt with in detail in the chapter devoted to QC files below.

StudioMdl

`StudioMdl.EXE` is the command-line program that actually creates the *Half-Life* model files. It takes the name of a QC file as a command parameter and uses the QC to find the relevant SMD and texture files. The program also provides feedback on the amount of memory used for the completed model’s geometry, textures, and animation data, which can be useful in debugging and optimizing your models.

The finished product: .MDL files

The final output the process is a *Half-Life* model (.MDL) file, which contains all of the model data: geometry, textures, animations, AI hooks, and so on. It’s ready to be called by relevant code.

Seeing your model in the game

If you want to test the model in the game environment, you can insert it into a game map by placing WorldCraft cycler entity where you want the model to appear. The cycler will allow you to see the model when you view it in the game -- in classic first-person-shooter manner, you browse through the model’s animations by shooting at the cycler.

If your monster/character code isn't working yet, you can also check the appearance of simple behaviors (such as walks or death animations) by placing the model in WorldCraft as a "Monster-Generic" entity – effectively an empty AI shell that only knows how to move, run away, and die when shot. Since monster-generics can also play scripted sequences you can test any animation by calling it from a scripted sequence – this is a good way to get around some of the distractions caused by the fact that cyclers will loop continuously.

Viewing models with HLMV

There are a number of shareware programs available on line which will enable you to inspect a *Half-Life* model without needing to run WorldCraft or the game engine. The SDK includes `HLMV.exe`, a shareware model viewer with several useful features including the ability to turn hide textures, to show bones, and to play back animations at varying speeds. For more information about HLMV see the appropriate folder in the SDK.

Modeling Workflow

Planning A New Model

Building and animating a model is an inherently complex process, and good planning will make the job much easier. While it's certainly possible to make a successful model on instinct alone, devoting a few minutes at the outset to getting a good grasp of the whole arc of the task at hand is a good investment.

Modeling for a real-time game such as *Half-Life* demands some important decisions be made before the first polygon is built. At the very minimum, you should have a rough idea of how important the new model is to your project, what kind of performance characteristics you want from the model, and how much memory you can devote to it. It's also a good idea to think about the organization of your source material, since StudioMdl will need to be told where the building blocks of the model are located.

How important is this model?

Knowing the role that a new model will play in your project is an important first step, because many other decisions flow from the relative priority you are according to this element of your game. Memory and artist time are finite resources and should be allocated with care.

A number of characteristics can influence the polygon and texture budgets assigned to your model. A monster that attacks in packs, like the *Half-Life* Houndeye, will almost certainly have a lower polygon budget than a model that appears on its own. Models that appear everywhere in the game, like the *Half-Life* scientist, may need to be more efficient with textures than a model that's only appearing in a handful of scenes. Remember that game performance is based on the total amount of polygons (and to a lesser extent, the number and size of textures in memory) at any given time. You will have to make trade-offs between the resources available for different elements if these will co-exist on screen or in memory at the same time. The *Half-Life* weapon view models – each of which had to be able to be on screen at any time – had less than 150k of texture memory.

Budgeting Polygons

The number of polygons in a model depends on several factors. You should have a clear idea of what your target machine is, and how many polygons it can deliver at a reasonable framerate. In *Half-Life* we assumed a 166mhz Pentium with 24mb of memory and a 1mb Voodoo I accelerator card. We found experimentally that in an "average" room we could display around 3,000 polygons at a "reasonable" framerate of 12-15 frames per second. However only the Gargantua, Nihilanth, Gonarch models came anywhere near this total because they all were intended to appear in very carefully designed situations. Most of the human-sized characters were between 500 to 750 polygons, and pack monsters like the Houndeye were even lower. Don't forget to include weapon viewmodels, gib models and so on in your on-screen budget before figuring out how many polygons you can allot to a particular model.

Today's machines are capable of pushing far more polygons, so these limits would be considerably larger if the game were to be made now. Nevertheless economy in polygon use is always a good idea.
We talk about a model's "polygon count" when discussing its efficiency, but in many ways the real determining factor is the number of vertices in the model. Since the vertices determine both the shape of the model and other factors, such as how a texture or smooth group will be placed it is really the vertices you should be thinking about optimizing. Every vertex in your model must be there to do one of three things:

- Establish a significant contour

  What "significant" means in the context of your model is ultimately a judgment call. You should decide if a contour is significant based on how often, how long, and how well a model will be seen in the game – you can skimp on the details on the top of a 20' tall monster's head if you'll never be looking down on him. The basic rule of thumb is to discard a vertex if the player is unlike to notice its absence. If the vertex does establish either the model's outline or a significant aspect of it's shading (if, for example, you need to convey that a particular object is concave or convex as opposed to flat) it makes the cut; otherwise it's not really necessary.

- Hold a texture coordinate

  Since textures are a property of the individual triangles, if you need a certain space to hold different textures you will need different polygons – and thus different vertices – to hold them. The same holds true if you are using parts of the same texture map with several different projections.

- Keep an animating object's shape

  Since your model will be animating, you will need vertices to keep the model's shape intact as its skeleton moves around. A pair of legs could be modeled as a simple tube if the knee were never bent; but in an animated model you would need a ring of vertices to maintain the shape of the calf and another for the thigh.

Obviously there is more art than science in making these judgments, and there may be situations in which no good answer can be easily found. However a good global understanding of how your model fits in to the project as a whole is the best starting point for making the necessary trade-offs.

Finally, you will also need to decide if the model will have multiple versions ("bodygroups" – the `$bodygroup` command in the QC file documentation, p. 44) such as the various weapons on the *Half-Life* soldiers or the swappable heads on the scientist model. Remember to include the polygons for such items in your budget.

Budgeting Textures

Budgeting textures is somewhat different from budgeting polygons. You do get some help from the fact that the *Half-Life* engine automatically shares texture information between multiple instances of the same model. This means that a whole screen of Houndeyes or Vortigaunts uses no more texture memory than a displaying a single individual. On the negative side, however, any time a model's data is accessed – even if it is not displayed on screen! – its textures will be loaded (for a partial solution to this problem see the sideboard on "External Textures" below). Moreover if you exceed the texture memory limits of a hardware 3-d accelerator the performance hit is much more obvious than the gradual slowdown imposed by polygon overload. In such a situation there is a perceptible hitch in game play rather than a general degradation in framerate. For this reason texture budgeting is, if anything, more critical to the success of a model than polygon counts.

Half-Life's target system assumed a Voodoo I accelerator card with only 1MB of texture memory. For this reason many *Half-Life* textures are extremely small. Even weapon view models, which are on-screen almost continuously, were limited to around 125k of texture memory – about equivalent to two 256 pixel square maps at 8 bits. Characters such as Barney (the security guard) used 256k, the equivalent of 512 x 512 bitmap.

Fortunately most contemporary accelerators have much larger budgets and accordingly higher texture resolutions. However the same basic problem of resource allocation remains. Moreover the rise of GL-based hardware acceleration has taken away a lot of freedom in your choice of texture sizes. OpenGL hardware generally wants textures to come in power-of-2 sizes, i.e. dimensions of 2, 4, 8, 16, 32, 64, 128, 256 etc. Many hardware cards have additional limitations as well. Older Voodoo-based cards will crash if given textures with an aspect ratio greater than 8:1 (e.g. A 64 x 8 texture is OK but 256 x 16 is not). Other cards want all of their textures to be squares. Unless you are absolutely certain about the hardware you'll be running on you should use only textures which are power-of-2 squares (see table).

External Textures

The model caching system, which will load an entire model to access any of its functions, can impose steep memory costs. If you have a model which you will need to access often without displaying it on screen, you can use the QC command `$externaltextures` to force the model to store the texture data separately from the rest of the model's data. This is often useful for optimizing texture usage. For more on external textures, see the QC command listing, p. 38

Powers-of-two textures

Texture size	Memory footprint for 8 bit texture
8 x 8	< 1k
16 x 16	< 1k
32 x 32	1 k
64 x 64	4 k
128 x 128	16 k
256 x 256	64 k
512 x 512	256k

As you can see from the table at left, this range of texture sizes does somewhat limit your ability to fine-tune the amount of texture memory allocated to a model.

It may help your decision process somewhat to remember that there is a conceptual trade-off between the complexity of geometry and the complexity of textures. A model with very stark polygon limitations will depend very heavily on its textures for character, visual interest and the appearance of greater detail. A model with more complex geometry may be able to use somewhat lower resolution maps if geometry can be made to provide the model with visual impact.

Planning for animation

Perhaps the most difficult element of a new model to plan is its animation. Generally speaking animation is the most difficult and time consuming part of the production process for models. Fortunately skeletal animation data is quite inexpensive to store compared to geometry and textures. For this reason your animation planning should concentrate on establishing a clear idea of your model's movement style and required animations. The things your model will be called on to do will dictate a great deal of how you construct it.

In a low polygon-count environment there are many arrangements of geometry that look good in one pose and terrible in another. Make sure that the modeler knows where and how much the character is intended to bend, because it is extremely difficult and time consuming to go back and change the geometry of a finished character to fix a movement problem.

Designing your skeletal structure

The skeleton of the model is the most important determinant of how it will animate. **All *Half-Life* animations are created by moving or rotating a bone.** The mesh itself is not deformed directly by the animator. This is a major change in methodology from older animation systems in which every pose was, in essence, a complete morph target where each vertex could be tweaked by hand. It is certainly possible to move individual vertices by attaching them each to their own individual bones in the skeleton – however from a practical standpoint this technique is cumbersome and should be used sparingly. It should also be obvious that the positioning of the skeleton will determine the point around which limbs pivot and how far the deformations from a movement will extend.

You can minimize time and effort by not creating a needlessly complex skeleton. Many modelers, for example, will use the Character Studio biped as the basis for their skeletons. Few low-poly characters will have individually modeled fingers or toes, however. It is a good idea to prune away the unneeded extremities so that they don't clutter the animator's workspace for no reason. Moreover attaching vertices to the skeletons in far more tedious if there are a large number of small bones in the skeleton. Finally, bear in mind that the animator's workload will be directly related to how easy it is to pose your model -- a skeleton whose every pose requires a dozen individual rotations will take much longer to animate than one which can be posed with a single IK movement; on the other hand Max's IK engine itself is somewhat unstable and needs careful planning to get good results. There are good examples of how to set up a skeleton in the Max r3 tutorial manual.

A note about scaling

The SMD Exporter recognizes only translation (movement) and rotation; it does not have any way of relaying changing scales to the engine. For this reason any change of scale between the animations and the reference file, or during an animation, will cause Bad Things to happen to your model in the game. If the model needs to be scaled you can apply global scales in the QC file. **For the sake of safety, its best never to use the scale tool on your skeleton** (it is OK to scale portions of a mesh while building the model, however, since this does not involve the skeleton). Since Max's *Mirror* tool uses scale transforms to do its work, the mirror tool is best avoided while making skeletons.

Skeletons and game engine data

The last point to consider when setting up your skeleton is what, if any, special functions your model will have. Models can have attachment points which are hooks for the placement of sprites, weapon models, or other in game entities on the model (see *Attachment points* for item, p.38). Every attachment point must be attached to a bone, and it's position in space is specified relative to the bone's coordinate system. If the coordinate system of the bone is skewed or hard to intuit, placing attachment points will be unnecessarily difficult.

Bones can also be directly controlled by the game engine rather than animated by keyframes; good examples of bone controllers are the jaws on the Barney and scientist models, which are moved by audio input rather than by animation data. If you are planning on controlling any aspect of your model in code you'll need to include bones for the code to use. For more information on bone controllers see *Controllers* on p. 41.

Setting up your work files

The final step in planning your model is establishing a working hierarchy of source directories. Consistency in file placement is extremely important because the QC files are the only method by which StudioMDL can find the source files it needs; moving and renaming files may make it impossible to rebuild a model when needed. The illustration on the next page contains a sample directory tree which shows you how to organize a modeling project.

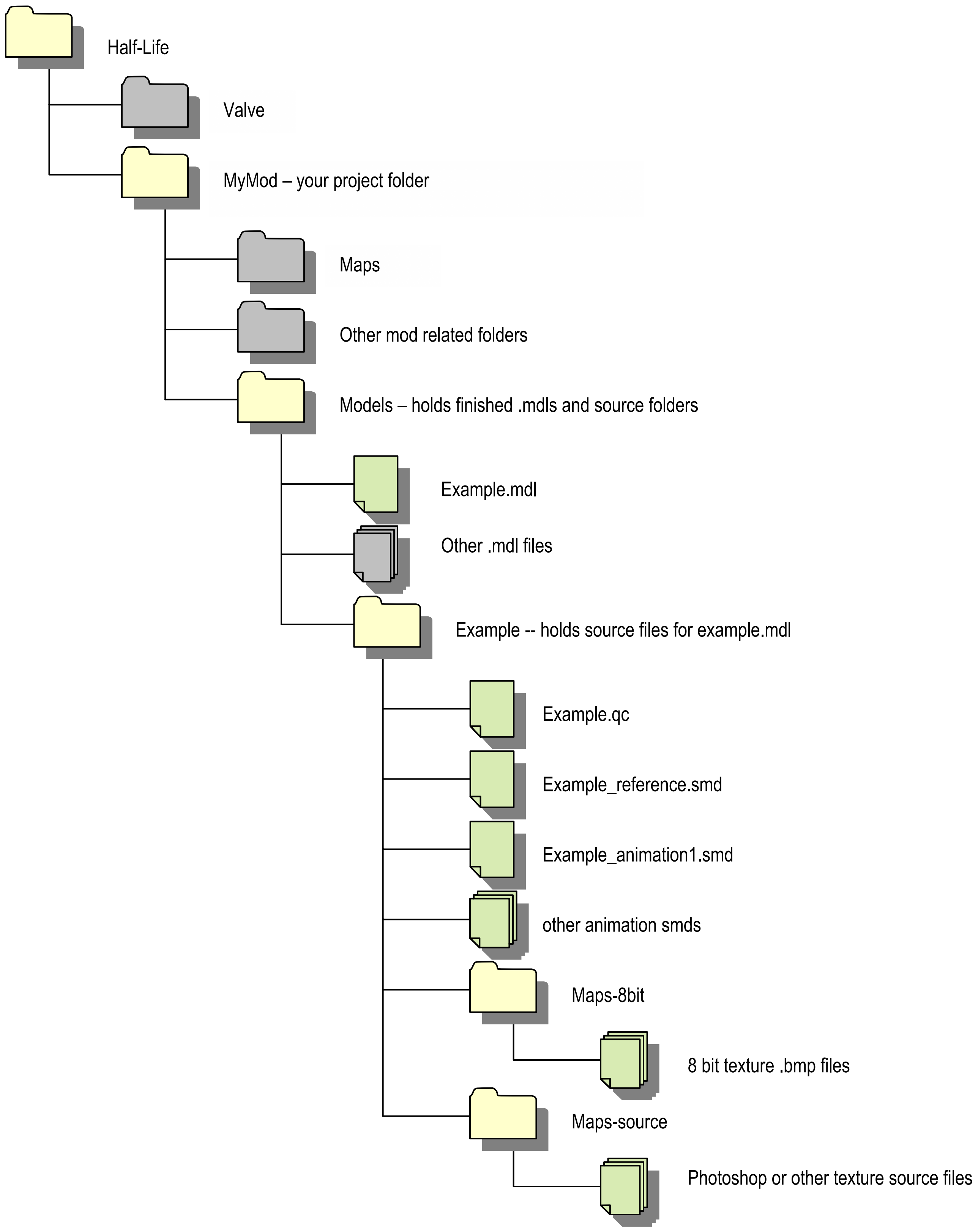
The advantage to this structure is that it allows you to standardize the location of resources for your team. That way you can call StudioMdl from the *Half-Life* directory and give it only relative directions to the QC files. If everybody on the team uses the same directory structure, anyone will be able to rebuild the model without worrying about different drive letters or naming conventions. This structure also makes it much easier to create and maintain a batch file for compiling QCs (see discussion of batch files on p.42).

It's also a good idea to establish firm naming conventions for your files. If you look at the *Half-Life* files on the CD you'll observe that many of them have confusing, non-descriptive names – we at Valve learned the hard way about the value of using clear and informative names for our work. You are free to adopt any system you choose, however – neither the exporter nor StudioMdl demand any particular naming structure to function. The only features which affect your file names are Chrome maps (see *Chrome Maps*, p*. 40*) and color-shifted multiplayer maps (see *Team colors*, p.40)

Finally, it's a good idea to plan on authoring your textures at somewhat larger sizes than you intend to use – if more texture memory becomes available it's difficult to scale up a finished texture to a larger size, whereas it is quite simple to make a scaled down copy. Moreover the exporter requires 8-bit textures, but most paint programs work best in 24 bits. For this reason you should keep separate copies of your 24-bit originals and your 8-bit final artwork. Its wise to keep the two in separate folders to avoid accidental overwrites.

The next page illustrates a sample directory tree structure that works well with *Half-Life* development tools.

Figure 1: sample directory tree structure


Half-Life
Valve
MyMod – your project folder
Maps
Other mod related folders
Models – holds finished .mdls and source folders
Example.mdl
Other .mdl files
Example -- holds source files for example.mdl
Example.qc
Example_reference.smd
Example_animation1.smd
other animation smds
Maps-8bit
8 bit texture .bmp files
Maps-source
Photoshop or other texture source files

Modeling Basics

Building a Mesh

The following sections assume that you're already familiar with the basics of the 3dStudio Max interface. If you're not familiar with any of the following concepts

- Editable Meshes
- Modifiers and the Modifier Stack
- Sub Objects
- Texture Maps
- Extrusion

You may want to see the appropriate sections of your Max documentation before continuing.

Strategies

Traditionally modelers have used two opposite approaches to modeling for real-time environments. Many modelers begin with a simple geometric primitive, such as a sphere or a box, and then add detail to the model by moving vertices, extruding faces, cutting or tessellating edges, and so on. Others prefer to model a fully detailed character with all of the various tools that Max offers – NURBS modeling, patches, CSG Booleans, and so forth – and then refine that model down to a lower polygon-count editable mesh.

Which strategy you adopt is partly a function what you need to achieve with your model, and partly a matter of personal preference. There is no reason why you cannot mix-and-match techniques from both approaches. Here's a quick overview of the pros and cons of the two different methods.

Modeling from the ground up

The biggest advantage to starting with a primitive and gradually reshaping it into a complete model is that you'll know exactly why each vertex is where it is. You can also keep a much tighter reign on poly budgets because you don't have the chance to become overly attached to a particularly nice feature that you just can't afford in polygon terms. Finally, you'll have the advantage of knowing that the entire mesh is a seamless network rather than a collection of intersecting, unconnected meshes that can be hard to understand and clean up.

One common problem with the completely hand-modeled approach is obvious – you'll be hand positioning every vertex in your model. This can be a burden, particularly if you need to make objects which are a compound of recognizable shapes – it's difficult to get a satisfying sphere or chamfered box by hand-editing alone. One must also work extra hard to avoid the tendency of extruded-and-scaled models to be overly symmetrical and blocky. A good rule of thumb is that no two vertices which do not *have* to be aligned should be – i.e., if you have a line of vertices defining the line of a character's back, and the back is not supposed to be rigidly straight, then you should make sure those vertices are not simply stacked straight above each other.

High-to-low strategy

Modelers who prefer to work "from the top down" like the fact that the high-end tools, such as NURBS and Booleans, give them the ability to conceptualize complex forms that would be difficult or impossible to build from the ground up. Another advantage is that more complex geometry and textures can be rendered out to form the basis of textures for less detailed models. Finally, top-down models are usually more impressive for promotional purposes.

On the down side, top-down modeling can be inefficient, because it involves doing more work than is absolutely necessary. Unfortunately the optimization tools provided by 3dStudio are generally not good enough for automatic reduction of a mesh down to a useful figure, so there will be a lot of work reducing the models you've already worked hard to build. There is also the danger of fixating on details which are appealing in themselves but too small or too complex for your polygon budget.

Importing geometry from other programs

This is a good place to point out that you can import models from a large variety of other 3d packages into 3dStudio Max . Max reads a number of formats including .DXF, which is supported by most modeling packages. One of our modelers at Valve uses Hash Studios' *Animation Master* to generate all his meshes before completing them in Max . Others use Max plug-in modeling tools such as MetaClay or DaVinci. If you are more comfortable modeling in another program, it is perfectly possible to use Max simply for the final preparation of the model. The simple rule to remember is that **any object that can be converted into a Max editable mesh can be textured, given a skeleton, and exported to *Half-Life***. Unfortunately there is no widely available mechanism for importing animation into Max at the present time, so you will probably have to do your animation work in Max regardless of where you create your geometry.

Max tools overview

However you make your meshes, there are a number of tools that impact directly on the way your model will interact with the SMD exporter.

Edit Mesh modifiers and the Editable Mesh base object

Since the SMD export process only supports *editable mesh* objects, most of your work will be done with Max's mesh tools. Max offers two ways to work on a mesh. Any editable mesh object offers a variety of tools for mesh editing, available in the command panel rollout.

Alternatively adding an *Edit Mesh* modifier to any object will effectively convert it to an editable mesh from that point in the Max stack onwards; *Edit Mesh* also offers a selection of mesh editing tools similar to (but not the same as) the tools in the Editable Mesh base object.

Because an *Edit Mesh* modifier effectively creates a copy of the entire object and then makes a mesh out of it, having multiple *Edit Mesh* modifiers on an object will eat up memory and slow performance on the model. For this reason it's a good idea to periodically collapse your stack rather than accumulating a mass of *Edit Mesh* operations.

Edit Mesh functions

You should be familiar with all of the basic tools for Mesh editing in Max . If you are not, review the Max documentation (starting on p. 640 in volume I of the Max R3 manuals or page 12-1 of the Max R2 User guide) before proceeding.

Here are some notes to help you use Max's common mesh editing tools in the *Half-Life* model-building process. All of these tools are found in the Modify panel when you have an editable mesh object selected for editing.

Collapse and Target Weld

These are two alternate ways of reducing the number of vertices.

The Collapse button will combine any number of selected vertices into one vertex. The new vertex will be located at the centroid of all the selected vertices.. The collapse button also works on faces and edges – it operates on all of the vertices in the selected face(s) or edge(s). When you collapse vertices or faces all the edges connected to the vertices or faces will be connected to the new, collapsed vertex. Collapse is particularly useful for simplifying meshes, particularly because it allows you to quickly delete large numbers of vertices or faces without breaking the continuity of your mesh.

The Target Weld button is important for more detail-oriented vertex reduction. When target weld is activated you can essentially force one or more vertices to merge with a "target" vertex at the target's location. Target weld is particularly useful for removing interior vertices from an area whose contours you want to preserve. You can use target weld to clean up your mesh by simply dragging unneeded vertices onto their neighbors.

Weld Selected

The Weld Selected tool will collapse selected vertices in a manner similar to the collapse tool. However rather than collecting all of these vertices into a single point, Weld Selected will collapse only meshes that fall within a specified tolerance. This is useful for stitching together vertices which are close to each other, either to eliminate features which are too small to see or to connect polygons which are adjacent in space but don't share vertices.

A very common time-saving tactic in modeling is to build half of a symmetrical object and then simply attach a mirrored copy of the completed half to make the whole model. Weld Selected is a simple, fast way to automatically join the mated edges along the seam.

Extrude

In earlier versions of Max (r1.2 and r2) extruding is one of the most common methods for modelers using a "ground up" modeling strategy. Since an extruded face (or group of faces) adds new faces along the depth

Issues with Max Mesh Objects

Every element in a Max Mesh object is stored in an indexed list. A mesh will have a list of vertices, a list of edges, and a list of faces, a list of materials, a list of texture coordinates, and a list of smoothing groups. If any one of these elements is changed or deleted, all the subsequent entries in the relevant list(s) must be renumbered to accommodate the change.

Why do we care? For two reasons. First, older versions of Max (r1.0 through r2.5) are prone to errors when rebuilding the lists. So deleting a single vertex may have the nasty side-effect of randomizing the texture coordinates, smoothing groups, or material ID's of the entire object. Anything which causes the vertex or other indices to be rebuilt can trigger this bug – common causes are vertex deletions, vertex welds or collapses, slicing faces and so on. Unfortunately there is no way to predict this problem before it occurs (although it's frequency does seem to have some relationship to the number of *Edit Mesh* operations in your stack) – the only safeguard is to use Max's autosave to keep backup versions of your file so you can revert back in case of problems. Sometimes deleting the offending *Edit Mesh* operation will cure the problem but in older versions, especially r1.2, this is not guaranteed.

The second troubling aspect to Max's handling of *Edit Mesh* information is that all the operations that make up an *Edit Mesh* modifier in your stack are also stored by index. Therefore if you change a mesh upstream from a particular *Edit Mesh* mod you will change the contents of the index lists – but the edit mesh mod will still be using the values from the old index. Effectively this means that the results of the *Edit Mesh* are now random. Fortunately Max will warn you before letting you open an *Edit Mesh* mod which might cause this to happen. Be very careful before changing meshes upstream in your stack!

The basic lesson of all this: don't allow yourself to accumulate a large stack of *Edit Mesh* operations. Collapse the stack whenever you're ready to lock in your mesh and prevent it from being corrupted of the extrude, extrusion is an easy way to create a new section in the mesh with regular subdivisions. To understand how, imagine extruding the top face of a cube: this would create a rectangular solid with two vertical subdivisions, and each subsequent extrusion on the same face would effectively stretch the solid and a new subdivision. With the advent of the Slice and Cut tools (see below) is now often easier to simply draw the subdivisions you want instead of extruding them out

Extruding a face with a depth value of 0 and then collapsing it is a fast way to create a vertex in the middle of an existing face.

Note that Extruded faces by default do not smooth to the faces around them. If you want your extruded area to smooth with the faces around it you'll have to apply the appropriate smoothing groups yourself.

Slice and Cut

The Slice and Cut tools are very useful for putting vertices and edges where you need them. When using these tools you'll have to pay careful attention to the new edges that the cutting operations generate. Max "Faces" are assemblies of triangles (supposedly co-planar triangles) arranged origami-fashion to create the multi-sided face. The edges that separate the triangles inside the Max face are usually invisible.

However the invisible edges do still control the way the mesh will look when it is deformed. There are many potential circumstance where your vertex assignments are correct, and your mesh's faces seem to be the correct, and yet two similar faces will deform differently. This is because the triangles making up those faces are arranged differently.

To correct this sort of problem, you may need to make those hidden edges visible to make sure that the triangles deform the way you want them to. The <u>Turn</u> <u>Edges</u> tool enables you to change the orientation of edges inside your faces.

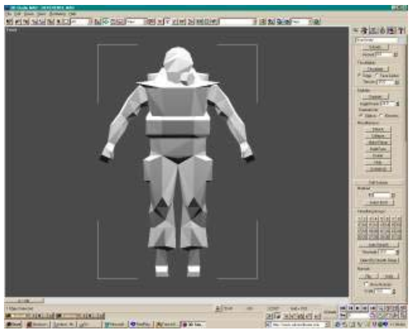
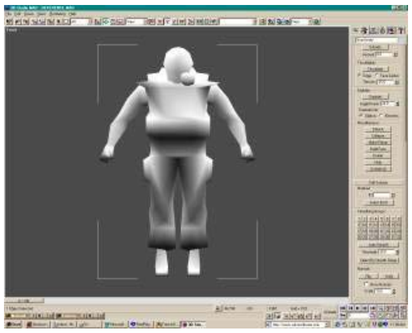
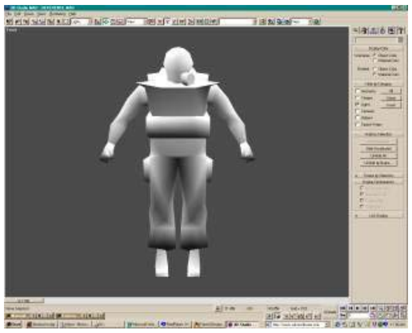
Smoothing groups

The *smoothing group* system is Max's mechanism for identifying which faces in a mesh are supposed to form part of a continuous contour. Two faces which belong to the same smoothing group and share an edge will appear to be part of a single, smooth surface; two adjacent faces which belong to different smoothing groups will show a sharp division or crease. Your *Half-Life* model will have the same shading as your Max model.

Checking smooth groups

You can verify that your smoothing groups have been preserved by looking at the model in the engine with the console command "r_fullbright 1" or by choosing the "smoothshaded" render option in *Half-Life*MV.exe.

Proper use of smoothing groups is critical for disguising the effects of low polygon counts; fairly crude geometry can be made to seem much more organic by adroit placement of highlights and smoothed edges. These illustrations show much smoothing groups help to clarify a model's form:

It is possible for a face to belong to multiple smoothing groups. If any of the smoothing groups assigned to adjacent faces are the same, the faces will smooth.

You can modify smoothing groups with the appropriate buttons in an editable mesh or *Edit Mesh* modifier, or by adding a *Smooth* modifier to your stack. This will apply to whatever selection you pass to it from the previous item in the stack. Note that *Smooth* is not the same as the *MeshSmooth* modifier, which will actually add vertices and faces to your geometry!

Two sided faces

The exporter supports only single-sided faces. To avoid surprises when exporting, make sure you test your model with single-sided faces. If you need a two-sided polygon you'll need to build two copies of the geometry with opposite normals.

Vertex Colors

At present the SMD exporter does not support Max's vertex color mechanism. If you are using vertex colors for any reason the vertex color information won't be exported to *Half-Life*.

Modifiers

You can use any of Max's modifiers to help you model. Bear in mind, however, that the finished product must be an editable mesh object (You can always convert to an editable mesh by collapsing your stack or applying an *Edit Mesh* modifier. Any special properties conferred by a modifier (such as the animated deformations in a *Path Deform* modifier) will be lost when the model is exported; however the changes to the geometry visible in your viewport should be accurately maintained in the exporter. To make sure that your Max scene is an accurate preview of your eventual model, it is wisest to convert your model to an editable mesh before attaching it to a skeleton or exporting it.

Here are some of the handiest modifier based tools for modeling:

FFD modifiers

The FFD or Free Form Deformation modifier is a useful way to shape an entire mesh at once. The FFD is particularly useful for molding primitive shapes such as spheres and cubes into the building blocks of an organic model. FFDs also duplicate many of the abilities of other Max modifiers such as *Taper* and *Twist* with a more readily understandable and controllable interface.

FFD's are very useful for creating forced perspectives on models which are unduly distorted by the standard 90 degree F.O.V. on the *Half-Life* camera, such as weapon view models (see below, p. 37)

Mesh Smooth and Tessellate

If you do work from the ground up, you can add complexity to your mesh quickly by using the *Mesh Smooth* and *Tessellate* modifiers. The helpfulness of *Mesh Smooth* modifiers is greatly increased when you use them on vertex or face sub-objects (in Max r3 and later you'll need to turn off the "apply to whole object" checkbox to work with sub-objects).

Optimize

For those working down from more complex models, Max's *Optimize* modifier has only limited utility for simplifying and reducing the polygon count of your mesh. It does a fairly good job of eliminating entirely redundant vertices and faces (for example, simplifying a plane with many unnecessary subdivisions). However the optimize tool has many drawbacks for more serious attempts at polygon reduction. It sometimes rebuilds the mesh incorrectly, resulting in garbled texture and materials mappings. It usually leaves a confusing spider web of edges, many of which will need to be hand tuned for good movement. Finally, the optimizer has an unfortunate tendency to destroy symmetries when used on a whole model.

Generally *Optimize* is useful for making a first pass on a detailed model or geometry that you have imported from another package. However you cannot count on it giving you a meaningful final result. Hand optimization, using the vertex welding and collapsing tools, is still the most reliable method of reducing your mesh.

Cap Holes

The *Cap Holes* modifier is often handy for filling in gaps created by mesh editing, or polygons that failed to import correctly from DXF files. However *Cap Holes* has a tendency to misplace the normals on faces – if you are working with backface culling, the flipped faces may not be visible at all if you are in a shaded view.

Cap Holes does not always make intelligent choices about the edges inside the faces in creates – particularly if you use Cap Holes in an area that will deform when animating, be sure to double check position of the edges.

Dealing with Normals

The SMD export attempts to preserve the surface normal information (i.e. the "facing") of the geometry you create in Max . Since the polygons in your final *Half-Life* model will be single-sided, make sure you work in Max with the "Backface Culling" option turned on to give you a correct preview. Changes you make to the model with the Normal modifier or the analogous tools in the editable mesh base object should be reflected in your final model.

If you need a two-sided polygon in your model for some reason, you'll need to make a separate face for each side of the polygon concerned. Make sure that you the two "sides" of your polygon aren't set to smooth with each other!

On rare occasions your model may be exported with all of its normals reversed. If this happens you can correct the problem in the QC file with the $reverse command (see the QC file documentation for details).

Compound Objects

If you model with compound objects, bear in mind that they will have to be converted to editable meshes at some point during your modeling process.

Lofts

Lofts are useful for a number of types of modeling. One of their most useful properties is their ability to generate mapping coordinates on objects which could not easily be textured with standard planar or cylindrical mappings. If part of your object is going to be a flexed tube which needs to have a texture that follows its contours – the ribbed hose on a gas pump, for example – a loft object (converted to an editable mesh, of course) is an excellent way to model the shape and to attach the texture coordinates (for more on texture coordinates see *UV Coordinates,* p. 20 below)

ShapeMerges

The *ShapeMerge* compound object lets you project a spline shape onto a mesh. The spline will cut edges into the mesh, creating new polygons and vertices. ShapeMerge is an excellent way to produce complicated shapes for extruding. You can even adjust the number of vertices and edges in the new shape by adjusting the "Steps" value in the originating spline shape – this number represents the number of line segments between each of the spline's control points when the spline is projected onto the mesh.

However you need to pay attention to the underlying geometry when you project the shape. Since the face onto which you are projecting will probably have hidden edges (see the discussion of invisible edges above) you will probably have a few unnecessary vertices and edges. These can easily be cleaned up using the target weld tool or other methods.

Booleans

Boolean compound objects are a very powerful tool for making complex geometry with multiple intersecting forms. Unfortunately the Boolean process will almost always produce a large number of unnecessary faces and vertices. These will have to be cleaned up by hand. Target Welding (above, p. 13) is a very useful method for eliminating the large number of unnecessary vertices which always result from Boolean operations.

Other Tools

You can create your meshes using any of Max's modeling tools – Nurbs surfaces, making a spline cage mesh with the "Surface" mod, lofts, extrusions and so on. Likewise, you can modify your models using a variety of Max modifiers such as bends, tapers, Free form deformations and so on. The only limitation is that the end result of your Max modeling efforts will be exported as a simple mesh; it will not retain any special properties conferred by a modifier (for example the "Flex" modifier's soft-body characteristics). For this reason you should consider collapsing your stack periodically to be sure that your object will behave and export in a predictable fashion. Generally speaking you always want to finish your mesh building process with a single editable mesh object in your stack – this will make subsequent operations much faster and more reliable, and it will insure that you have WYSIWYG connection between the mesh in your Max scenes and your *Half-Life* model.

Texturing Basics

Once you have completed your mesh, you'll want start adding textures to your model. Texturing a model is usually an iterative process in which you paint a map, apply it to your model, and then adjust both the map and its application to the model to fine tune your effect.

Texturing for the *Half-Life* engine uses only a subset of the tools for texturing in Max . *Half-Life* models support only bitmap textures – Max's procedural textures, such as the "Checkers" or "Perlin Noise" maps are not available in *Half-Life*. *Half-Life* models also have only a single mapping channel for each face – where Max allows you to apply maps representing a model's specularity, transparency, and so on in *Half-Life* you can map only the model's base color (corresponding to the "diffuse" channel in a Max material.

Half-Life models also support only one set of texture coordinates, corresponding to Max's Map channel 1 (since the *Half-Life* models will never have more than one texture map on a particular face, extra map channels aren't necessary).

Before continuing, you should be familiar with these basic Max concepts:

- *Materials*
- *Texture Maps*
- *Bitmap textures*
- *Gizmos*

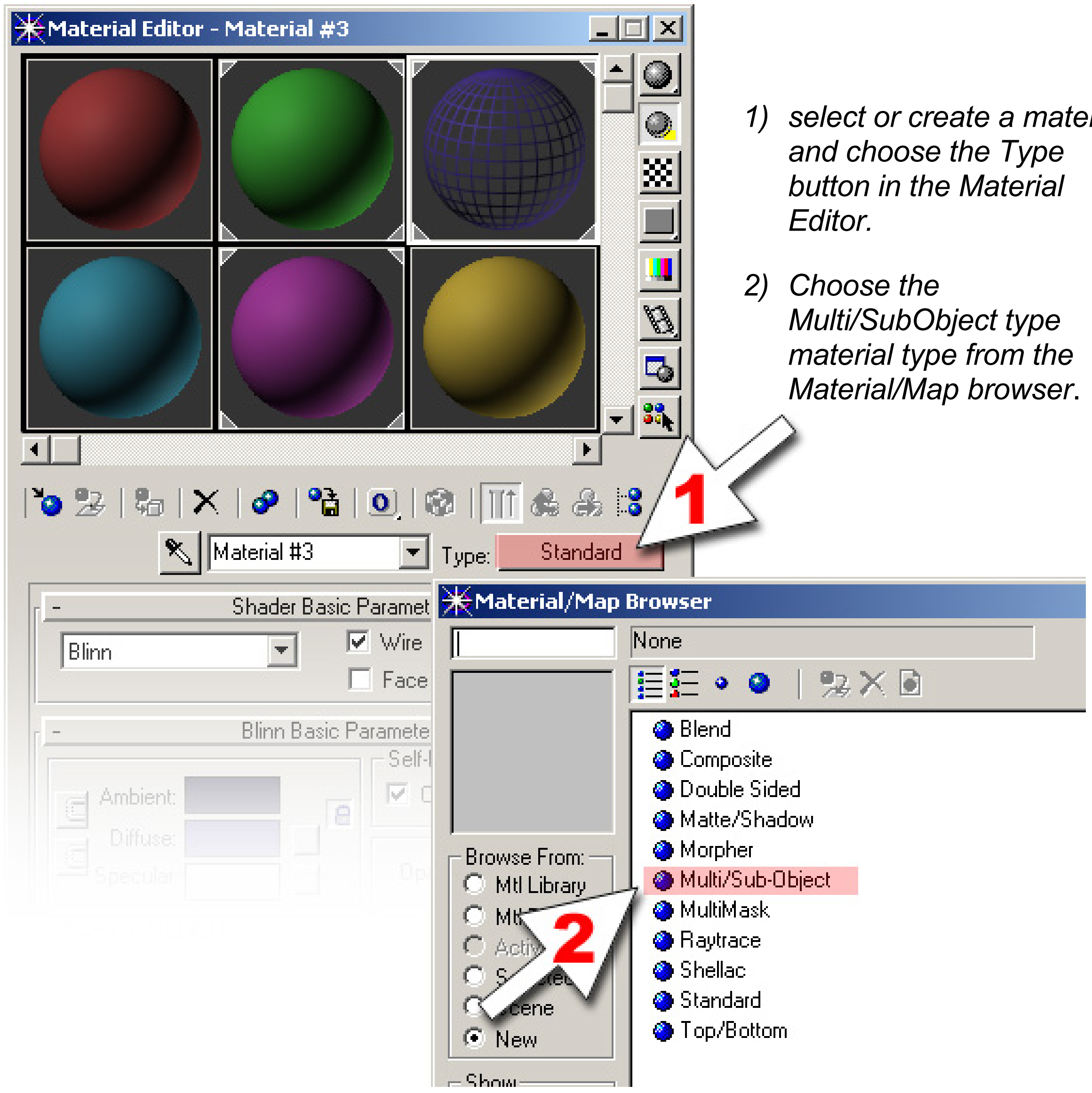
Setting Max Materials

To texture your model, you'll first need to create a material in Max . Because Max will not allow a single mesh object (remember that your final *Half-Life* model will consist of a mesh and a nonrendering skeleton) to have more than one material applied to it, you'll need to create a *Multi/SubObject* material.

Multi/SubObject materials are essentially containers with some number of sub-materials, each of which works like a normal Max material of the "Standard" type. Technically, each submaterial is a complete Max material with controls for specularity, opacity, bump maps, etc. For our purposes, however, each submaterial in your new Multi/SubObject is really just a container for a texture map, which you will assign to the submaterial's Diffuse channel.

You create a Multi/SubObject material by opening a Max material and using the "type" button (to the right of the dropdown box with the Material's name) to choose the Multi/Subobject type:

Figure 2: Creating a Multi/SubObject Material



1) *select or create a material and choose the Type button in the Material Editor.*

2) *Choose the Multi/SubObject type material type from the Material/Map browser.*

The resulting material will appear in the Material editor as a list of submaterials and a spinner for setting how many submaterials to use. You should make the number of submaterials equal to the number of textures of the textures you intend to apply to your model. Note that you will need to use a Multi/SubObject material even if your model contains only one texture – it will simply be a Multi/SubObject material with only one submaterial.

It's a good practice to give each submaterial a unique name and also to set its diffuse channel to a unique color; this will help you see more easily which faces use each material, and which material corresponds to which texture.

Applying Material ID's

Once you have applied the new Multi/SubObject material to your mesh, you will need to specify which faces use each submaterial. This is done by selecting the faces in an *Edit Mesh* modifier (or the editable mesh base object) and setting their material ID numbers – these correspond to the numbers of the submaterials in your Multi/SubObject material.

You can also apply material ID's by selecting faces in a *Mesh Select* modifier and then applying a *Material* modifier to them.

If you have set each submaterial in the Multi/SubObject material to a unique color, the end result will be a mesh with a patchwork of colored faces – each color corresponding to one of your intended texture maps. When you assign your textures to the diffuse channel of the submaterials, you'll see the colored faces replaced by your textures.

Previewing with Max Materials

To ensure that your Max scene matches the way your model looks in the engine, make sure that every sub-material applied to your mesh has the following settings:

The **Shader Type** should be *BLINN*

Ambient: should be white (rgb 255 255 255)

Specular should be black (rgb 0 0 0)

Specular level, **glossiness**, and **Soften** should all be 0

Opacity should be 100

Wire, **Face Map**, **2-sided** and **Faceted** should all be off (unchecked)

Painting your maps

You can generate your textures in any bitmap painting program such as Photoshop or Painter. *Half-Life* model textures have only to meet these requirements:

- *The texture is an 8-bit .BMP file*
- *Maximum size of the texture is no more than 1024 pixels in any direction.*
- *The texture must be stored a directory in your QC file's $Cdtexture list (see p.45)*

Any texture that meets these requirements is a valid *Half-Life* model texture. However there are some important considerations relating to efficiency that you should keep in mind when painting textures.

Palettes

Half-Life models use a 16-bit color space, even though each individual texture is only 8 bits. You can use any combination of 8 bit textures on your model, even though they have different palettes. This is a useful way to extend the color range of your model; you can put all of your flesh textures (for example) into one map and your clothing textures into another, which will give each texture much finer color resolution than combining all of these areas into one palette.

Some hardware accelerator cards (notably older Voodoo based accelerators) impose a slight speed penalty for converting each individual palette to the final 16-bit model. Thus a model containing two maps that have identical palettes is somewhat faster than the same model using two maps with different palettes.

Palette index numbers are only important if you are making multiplayer models with team colors (see p. 40). Textures that do not support color remapping have no other limitation on their palettes.

Texture Sizes

As noted above, most hardware accelerators prefer textures whose pixel dimensions are powers of two in both dimensions (see p. 6).

Number of maps

All other things being equal a smaller number of maps is faster than the same number of pixels taken from several smaller maps (i.e. one 128 x 128 texture is faster than four 64 x 64 textures).

For all of these reasons it's common to combine textures from different parts of the model in a single map (e.g., a map with the face and hands and another with both the front and back of the body). You should plan your use of texture maps with all of these considerations in mind.

Applying textures

You are now ready to apply your textures to the model. Go through the list of submaterials in your Multi/SubObject material and assign the appropriate bitmaps to the diffuse channel of the submaterials.

When assigning bitmaps, do not use the wrapping controls in the "coordinates" rollout. **None of the placement, tiling, or color adjustment controls in the Material Editor bitmap window are recognized by the exporter**. The only way to control texture placement is by manipulating the UVW coordinates on the mesh with a *UVW Map* or *UVW Unwrap* modifier. If you need to adjust the colors or contrast ratio of your bitmap you will have to do it in your paint program or by using the QC file's $gamma command (see p. 42). Do not use the color adjustment controls in the bitmap rollout. **The bitmaps rollout should be used only to specify the bitmap file**.

UV mapping

Max's basic mechanism for sticking textures to models is the *UVW Map* modifier. *UVW Map* gives you a moveable, re-sizeable gizmo with which you can interactively attach your texture to a model. The SMD exporter supports only bitmaps applied with UV maps; other types of mapping, such as environment mapping, or Max's "From World XYZ" and so on are not supported by the exporter and will produce unpredictable results if used.

How UVW maps work

The *UVW Map* gizmo represents your texture as if it were an object in the scene. The texture is projected through the "surface" of the gizmo as if it were a slide. Unlike a real slide, however, the projection is non-directional – it passes both ways through the gizmo, perpendicular to the surface. In a simple *planar* projection, the rectangular gizmo represents the texture – its four sides correspond to the four sides of the texture. You can imagine a *cylindrical* projection as a plane with its two sides curved around to meet each other. A *spherical* projection works like a cylindrical projection whose top and bottom edges have been tapered down to zero length, while its circumference at the center is unchanged.

In all types of projection, the *UVW map* modifier will assign two coordinates to each vertex in the selected regions of your mesh, representing the relationship between the vertex and the texture. The width and height of the texture are represented by values between 0 and 1; thus the coordinate

(0,0) would represent the top left corner of a texture while (1,1) would indicate the lower right corner and (.5, .5) would be the center. These coordinates are called U (width) and V (height), to distinguish them from X, Y, and Z.. U or V coordinates outside the range of 0 – 1 represent repeats of the same texture: UV (1.2,1.2) is the same as UV (.2,.2).

UV coordinates are the only way in which the exporter and StudioMdl know how to attach a texture to a mesh. This is why the repeat and tile values in Max's bitmap texture window are ignored. Since UV coordinates are only a function of vertices, all textures assignments must begin and end at vertices – you cannot make a texture projection stop in between vertices.

Since UV coordinates are relative -- UV (.5,.5) is always the center of a map, regardless of the map's pixel dimensions – it is possible to change the resolution of your maps after establishing a projection without changing the mapping; you can recompile a *Half-Life* model from SMDs without retexturing in Max .

Working with UVW Map

You should try to minimize the number of places in your model where the texture is projected onto the mesh at an oblique angle. This results in stretched pixels and an unpleasant smearing effect. Try to avoid projections that are close to parallel to the models surface – these produce the most extreme smearing. You can disguise unavoidable smears by lowering the contrast in affected areas of the texture – in a murky part of the texture the stretched pixels are much less offensive.

Clever use of UV mapping can help you save texture memory and time. For example, if you map a character's boots with a planar projection from the side, you can texture both feet at once with the same pixels, whereas trying to use one projection from the front would use a larger texture and waste the pixels in between the feet. By scaling and rotating the mapping gizmo you can mirror or skew textures to achieve – or avoid – symmetry. Unimportant areas of the model – such as the soles of the shoes – can be textured with small portions of re-used texture (a small area of a belt or a jacket often makes an unobtrusive sole when blown up larger)

Using multiple UVW Map modifiers.

Most models will require several UVW Map modifiers to establish all of your texturing coordinates. In a simple example, you might select the faces which make up your character's head and apply a Cylindrical UVW map to it. You can then make a new selection with a "Mesh Select" or "Edit Mesh" modifier and then add a new UVW map modifier to establish a second mapping for the torso, then another for the legs, and so on.

New UVW mappings will replace older ones where the selections overlap. The latest map always takes precedence.

Repeating Textures

You may be familiar with systems that use repeating or tiling textures. The SMD exporter does not support the tiling options in the Bitmap material window. If you want to make a repeating texture, you have three options.

If you are only interested in the look of the texture, and don't particularly need the extra efficiency of re-using your texture pixels, the simplest method is simply to include the repeats in your bitmap.

You can generate repeats by using UV coordinates outside the range 0-1, either using the Tile spinners in the UVW Map modifier rollout or by hand editing UV values in UVW Unwrap. If you use this method StudioMdl will automatically alter your texture to include the repeats; this is effectively the same as option (1). Your paint program probably uses a more sophisticated texture filtering method than StudioMDL, however, so consider fine-tuning the UV coordinates and painting the repeats into the texture yourself.

The most efficient option is to apply a number of individual UVWmaps, each of which constitutes a single repeat. If you have difficulty getting an exact repeat, UWVunwrap is an excellent way to tweak your mappings. The only limitation in this method is that each repeat must begin and end on discreet faces – you cannot end a repeat in the middle of a face, since only vertices can hold UV coordinates.

UVW unwrap

An excellent new tool for inspecting the UVW coordinates of your model is the "UVW Unwrap" modifier. This modifier is available in Max r2.5 and higher.

UVW Unwrap displays window showing your mesh in UV space. The Unwrap window corresponds to your texture map. You can choose to show your bitmap as a backdrop to make the relationship between model and map even clearer. The position of each face and vertex in the unwrap window corresponds to its UV coordinates. You can adjust the UV mapping of a vertex by moving it in the unwrap window (UVW Unwrap provides tools analogous to the standard move, rotate and scale tools for adjusting these vertex positions). You can use this feature to fine-tune the relation between the model and the texture – for example, by dragging the vertices for your character's nose to be exactly half-way between the eyes in your map or by making sure that a map edge's alignment to geometry is pixel-accurate.

If you're puzzled as to how to texture faces with cylindrical or spherical maps, *UVW Unwrap* is an excellent way to see how the mesh relates to the texture

Other texture mapping methods

Many Max objects offer you the ability to generate mapping coordinates when you create them. Since the coordinates are built with the objects, Max can assign mappings intelligently based on the objects' shapes, instead of forcing them into a generic projection plane, cylinder, or sphere. If you want a texture which visibly follows the contour of your surface (a checkered tablecloth draped over a table, for example), see if you can build your mesh using a tool that offers a built-in mapping option. If you create an object with the "generate mapping coordinates" option enabled, the object's texture coordinates will be retained when you convert it to an editable mesh and/or attach it to another mesh. You will have to combine all such objects into editable meshes before exporting them.

You can always erase mapping coordinates by adding a new *UVW Map* modifier.

If you use 3d party plug-in texturing tools beyond UVW Map and UVWunwrap you may not get predictable results. If your tool actually resets the texture coordinates of the mesh, it will export properly. The easiest way to assure yourself that your texturing tools will export correctly is to add a UVW map modifier to the model to check your results. An example of a plug-in which correctly updates UV mappings and will export properly is "Texturizer" from Sven Technologies Inc. Texturizer was used in the production of a number of the original Half-Life models (see

Easy Texture Alignment

Here's a useful trick for getting perfect alignment between your textures and your models.

Rotate your viewport to display the part of the model you want to texture. Select the faces you want to texture. Apply a *UVW Map* modifier of the planar type.

Align the *UVW Map* gizmo to the viewport using the "align to viewport" button. The "Fit" button will scale the gizmo to the dimensions of the face(s) you're texturing.

Use the windows ALT + Print Screen command to take a screenshot of the viewport.

Open your paint program and paste the screenshot into a new window. Crop the window to the size of the UVWMap gizmo in the screenshot.

You can now paint over the screenshot using it as a guide to registering the new texture with your model. If you need to resize to a powers-of-two resolution or square shape, you can do it after you have completed the texture. As long as you don't change the position or scale of the map gizmo, the texture and the model should be perfectly aligned.

Working with Half-Life Content from the Half-Life SDK, p. 59.).

Creating a skeleton

Overview

One of *Half-Life*'s key features is its skeletal animation system. As we've noted above, *Half-Life* stores a models animations by recording the movement and rotation of "bones" – entities which represent the supporting structure of the mesh. The most obvious illustration is a humanoid figure, where the "skeleton" corresponds pretty closely to an actual skeleton – a "bone" for the shoulder, a bone for the upper arm, a bone for the forearm, and so on.

There are three great advantages to the skeletal system, as opposed to many other systems which store each frame of animation as set of vertex positions. First, the skeletal system stores a lot less data – instead of evaluating every vertex position on each frame, the system has to store only the positions of a much smaller number of bones. Moreover the combination of position and rotation makes for easily interpolated values which can be played back at any framerate – the in-between frames in a skeletal animation will be correct, where vertex animations (which simulate rotation by moving vertices along curved paths) may interpolate strangely at framerates other than the one for which they were designed. Finally, skeletal animations can pass data to the engine about the position and orientation of things like sprites, hit-detection boxes, and so on.

The major drawback to the skeletal animation system as it is implemented in *Half-Life* is that each vertex is rigidly attached to one, and only one bone. Stretching and flexing can occur only on faces which have vertexes attached to different bones (for example, in the area of a knee where some vertices will be attached to the thigh and others to the calf). This means that some kinds of deformation, such as breathing, can be very hard to simulate with a skeletal system. The rigidity of the vertex attachments also means that in areas where very marked changes in position and rotation can occur, such as the shoulder, an unattractive twisting or folding effect can be seen. Good planning and repeated testing are the best ways to minimize this problem.

Almost all HL models will need a skeleton. Very simple models that do not need to change their shapes (such as a box of ammunition, a crowbar, or a chair) can be exported and animated directly without skeletons. If you need to control the movement of any individual vertex relative to the rest of the model, you will need a skeleton.

What is a bone?

In general terms, almost any Max object can be a bone for the *Half-Life* animation system. Max supplies an object type called "bones," which can be used in the skeleton of a *Half-Life* model, but Max bones are only one of the many kinds of objects which can be used as "bones" for our purposes. Almost any objects – dummies, grids, spline shapes, even lights or cameras can be part of a *Half-Life* skeleton, so long as they are linked together in a hierarchy Editable spline objects can be particularly useful because your animators can give them meaningful shapes – such a directional arrow – to make them easier to work with. Max's bones objects, by contrast, don't communicate directional information very well. Of course, cameras, lights and other special objects will not export any properties but their positions and rotations. However the positions and rotations can be animated and vertices can be assigned to them just like any other object.

It's also worth noting that almost any method of moving and rotating the bones in your skeleton will be supported by the SMD exporter. Any Max animation controller type that evaluates properly in your scene should produce proper animation data when exported.

Building a Skeleton

You create a skeleton by placing bones into your scene and linking them together to form a single hierarchy. The position of the bones' pivots will establish the center of rotation for the various parts of your model, i.e. the leg will rotate around the pivot of the hip, the arm around the shoulder, and so forth. Only the position and rotation of the joints are truly important in building your skeleton – the shape, scale, and other properties of the bone (for example fact that one bone is a Max Dummy object and another a Max Bone System object) are unimportant. Because only the position and rotation of bones are reported bones should not be scaled once they have been attached to a mesh. If you use scales in the construction of you skeleton you will have problems if the scale values change for any reason. For this reason it is best to avoid the scale tool altogether and create your skeleton with translation and rotation tools alone.

Some tools, such as the Bone System tool, create a set of objects already linked into a hierarchy. It is perfectly acceptable to use Max's link tool to join these groupings into larger assemblies. **Do not use Max's Group tool**. Group information is not exported and the results can be unpredictable. If you're used to programs such as Alias or SoftImage, where "grouping" creates a hierarchical linkage this may be confusing; however it is safest to avoid Max's group tool completely when modeling for Half-Life – use only the Link tool to create your hierarchies.

Remember that the skeleton cannot change between the reference SMD and the animation SMDs. You'll need to think about all of the motions you intend to create before you finalize your skeleton --- you cannot add, delete, re-order or re-name bones in an animation SMD. If you make changes in your reference skeleton you'll need to make the same changes to any completed animation SMDs as well.

For the sake of neatness, it's a good idea to make sure that your mesh object is linked to the root of your skeleton. *Physique* will offer to make this link for you automatically (see *Using* , p. 26).

Terminal objects

If you are planning on building a model with either the Physique modifier or the Skin modifier as your tool for attaching vertices to bones, you will need to add one extra object after the last joint of each limb of your skeleton. The vertex assignment tools need a "parent" and a "child" object to make a suitable attachment for the vertices. Don't bother animating any of these terminal objects – the movement of the dependent object is ignored (since it has not children, no vertices can be assigned to it). Max Dummies or even Max Point objects make good terminal objects, since they can be easily hidden or ignored while working.

Local orientations

When you create an object to be part of your skeleton you should be aware of its orientation. Bone controllers and attachment points are specified in terms of local orientations and can be extremely difficult to work with if those orientations aren't clear. Many animation methods, such as the EulerXYZ controllers, expressions, and IK chains, need to know which way an object is facing so that local rotations or movements can be calculated. You might want an elbow joint, for example, to bend only along one axis and the elbow object's orientation in order to restrict the unwanted rotations.

Generally Max creates objects so that their local Z axis points out of the viewport in which they are created; if you want your objects to have local axes which are the same as the world's X,Y, and Z axes you should create the objects in the top viewport. You may want to build different portions of the skeleton in different windows to establish your desired local orientations. Particularly if you are working with a system of Max bone objects it can be difficult to set up consistent orientations any other way.

To check the local orientation of an existing bone, simply select it and set the active coordinate system dropdown box (to the right of the Move, Rotate and Scale tools) to “local”. You can use the pivot in the hierarchy panel to change the local axes of the pivots when building the skeleton, however it’s not advisable to make any changes to the local axes after the skeleton is complete.

Link length

The Physique modifier will not recognize bone links between objects that are exactly in the same place, or below a pair of such collocated objects. If you need to bones to occupy the same space -- say you wish to have two different controllers to animate different components of an object’s rotation -- you will have to offset them by a very small distance (around .001 in world units seems to be enough)

Character Studio Bipeds

The *Biped* plug-in portion of Character Studio is an excellent way to start a skeleton for humanoid character. In addition to being a ready made template of a standard human skeletal structure the Biped offers a much more stable and reliable IK solution than most Max bone systems. Moreover the biped can be animated with inverse and forward kinematics at the same time: a very useful capability. Bipeds can also be used for non-human characters – see the examples in the Character Studio documentation for some good examples.

You can extend the usefulness of Bipeds by attaching other bone objects to them. You can add a second set of arms, wings, or other appendages to a biped simply by using Max’s Link tool.

Biped objects are the only exception to the rule about not using the scale tool to build or edit skeletons – when editing a biped in Figure Mode (see the character studio docs for more detail) it is acceptable to use the scale tools. The overall size of the biped should be scaled using the “height” parameter in the Biped Structure rollout. Using the scale tools to distort and reshape the Biped can help you in working with non-human, non-bipedal creatures as well. If you browse through the Half-Life content in the SDK you’ll notice that a number of non-humanoid creatures were created using Bipeds.

Updating and Editing Skeletons

If you find it necessary to make any changes to the your skeletal structure during the course of your project, you will have to update all of the animations which reference that skeleton. Every animation SMD will be checked against the skeleton in the reference SMD and any differences between the skeleton – adding, deleting or re-ordering or renaming bones – will cause prevent StudioMdl from correctly compiling the model.

If you need to animate bones which form a hierarchy for the purposes of exporting but need to behave like separate unrelated objects, you can force your bones not to inherit transformations or rotations from their parents – in other words, they will act as if they were not connected. In the “Link info!” section of the Hierarchy panel there are checkboxes that control your objects’ inheritance of translations, rotations, and scales in each axis. Turning inheritance off with these checkboxes will make the objects behave as if they were unattached, while preserving the hierarchical relationship for organizational purposes.

Mismatched Skeletons

One common cause of failed compiles is accidental changes to your scene’s hierarchy. If you add an object to your scene – say a moving dummy to help you time a motion in your animation – the SMD exporter will export the object along with your model. StudioMDL will see the skeleton of your model from the reference SMD and the skeleton from the animation SMD don’t match and will abort the compile. Invisible or oddly placed objects are the most common cause of the skeleton mismatch errors. Be sure to check for extra objects lying around in your scene if StudioMdl won’t compile.

Attaching Meshes to the skeleton

Once your skeleton is complete, you'll need to establish the relationship between the mesh and the skeleton. This can be done in one of two ways: If the model is not intended to change shape it can be linked directly to the skeleton with the link tool. If the vertices of the mesh are going to move, on the other hand, you'll need to apply a modifier to the mesh to specify how the vertices relate to the bones.

Linking Meshes

Simple objects can be attached to the skeleton simply by linking them to the bone that controls their movement.

Deformable Meshes

Perhaps the most time consuming part of the entire *Half-Life* modeling process is establishing the connection between the vertices of your mesh and the bones that you will use to animate the model. There are three ways to attach a mesh to a bone hierarchy, however before we look at each method individually there are two useful rules to remember:

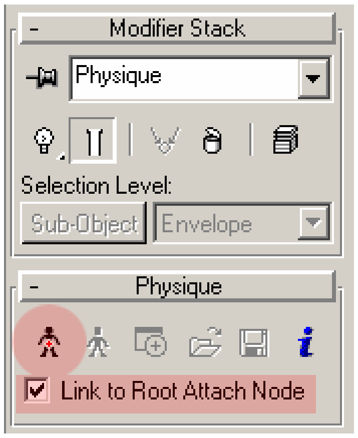
Make sure your vertices are attached rigidly. Some methods of attaching vertices offer the option of using deformable or weighted attachments in which a vertex's position is influenced by more than one bone. Half-life, on the other hand, will allow a vertex to be attached to only one bone. If you use non-rigid attachment methods your Max model will almost certainly not match the final exported model if you use any non-rigid attachments.

Attach your model to the skeleton after it is done! It may be useful to test a partially completed model for proper bending, etc. However it is wisest to attempt the final attachment only after you are confident that the mesh is in a finalized state. Mesh edits which come after an attachment modifier in the stack have unpredictable results. It is possible to adjust texture coordinates, material ID's, and smoothing groups after attachments. Anything which would change the topology of the mesh, however -- adding, deleting or moving a vertex or face – will almost certainly fail to export as expected.

Using ***Physique***

All of the models that shipped with *Half-Life* and TFC used the CharacterStudio "physique" mod to attach their meshes to their skeletons. Physique is the only method of vertex attachment supported in Max versions 1 and 2. For more detail on how to apply and work with the Physique modifier, see your Character Studio documentation.

First, assign the Physique modifier to your mesh. Make sure that you are assigning it to the whole mesh, rather than a vertex or face sub-selection. Be sure to check the "Link to root node" box so that your mesh will be linked to the skeleton's root. Press the "Attach to node button" and choose the root of your skeleton when prompted. If you are using a *Biped* skeleton object, the root node of the skeleton is the diamond-shaped center-of-mass object.



The Attach to Node button in the Physique modifier

You'll observe that the skeleton now displays a series of orange lines outlining the skeleton. If you some parts of your skeleton seems to be missing, you may have picked the wrong node as the root or you may have zero-length links (see *Link Lengths* above).

The basic methodology for assigning vertices to bones is

- *Switch to the "Vertex" sub-object level.*

- *Activate the “Select” button and select the vertices you want to assign.*
- *Activate the “Assign to Link” button and click on the orange line representing the bone link to which you want to attach the vertices.*

You will repeat these steps until all of your vertices have been assigned to bones. Remember that the vertices will move and rotate with the “top” of the bone – the joint on the link which is closer to the root in the hierarchy – so if you wish vertices to move with a knee, for example, you click on the link between the knee and the ankle.

Assigning Vertices With Envelopes

Physique allows you to assign vertices in large numbers quickly with “envelopes”. If you’re familiar with the envelope method, you can use it as a quick first pass at vertex assignment. If you do use envelopes, make sure that you use the “Rigid” envelopes to make your assignments. You’ll have to do some vertex-level assignments to resolve areas where the envelopes overlap. Vertices assigned with envelopes should work in the engine identically to vertices assigned by hand.

For more information about using envelopes see the Character Studio documentation.

Physique supports three kinds of assignment, which are represented by three colored crosses in the physique rollout. When you select vertices, the highlighted crosses act as filters – only vertices of the highlighted types will be selected. When you assign vertices, be sure to highlight only the green (rigid) crosses – as already noted the exporter supports only rigid vertex assignments.

You can test your assignments by repositioning your skeleton – the mesh will deform according to your assignments and you can easily spot mis-assigned vertices. You can assign vertices with your skeleton in any position – if you need to check your position against the skeleton’s original pose with the “Initial Skeletal Pose” checkbox.

For best results you should make sure that the *Physique* modifier is the last modifier in your stack. You can apply UV mapping, change smoothing groups, and change material ID’s after the *Physique* if necessary, but any mesh editing operations that take place after the *Physique* may be ignored by the exporter.

By default the orange link lines which Character Studio uses to represent the skeleton are spline curves – Character Studio uses this curvature to make softer deformations. This can interfere with the accuracy of your previews because this softening is ignored by the exporter. If your compiled model does not seem to match well with the Max model as you move and pose it, you can straighten out the skeletal curves. To adjust the curvature of a link, go to the “Link” sub-object level, select the link(s) you wish to change, and set the Tension value to 0.

Changing a mesh

Physique vertex assignments are a function of the topology of the mesh. If you make any changes to the mesh upstream in the stack from your physique modifier, your vertex assignments will be lost or confused.

If you discover that you will have to change your mesh after you have already assigned vertices, you may be able to use *Physique’s* “Lock Vertices” feature (available in the Vertex sub-object level) to preserve your assignments. Before making changes to the mesh select all of the assigned vertices and select the “Lock vertices” button. When you make upstream changes, new or moved vertices will copy the assignments of the original locked vertices. This can save a lot of time. It’s still always a good idea to complete your mesh before attaching it to a skeleton.

*Using **Skin***

With version 3.0 Max introduced the *Skin* modifier, which duplicates much of the functionality of *Physique* but which does not require Character Studio to operate. The SMD export script included with this SDK supports the *Skin* modifier as an alternative method of vertex assignment for modelers without access to Character Studio. For more information about *Skin* see the documentation on page 842 of the 3D Studio Max manual vol. I.

Using *Skin* is very similar to using *Physique*. Before applying the *Skin* modifier you should link the mesh object to the root of the skeleton with the link tool. Then hit the "Add bone" button and use the resulting dialog box to add all of the bones in your skeleton. You'll notice that the root bone will be missing from the list – in contrast to *Physique*, *Skin* remembers vertex assignments by the "bottom" of the link – i.e., when you make an assignment to the knee bone you're working with the link from the hip to the knee rather than the knee to the ankle.

To assign vertices with Skin, follow these steps.

- *Choose the "Envelope" sub-object level.*
- *Select the vertices you want to assign. They'll be highlighted with white squares.*
- *Use the "Reset Selected Vertices" button to clear their assignments (this button is located at the very bottom of the rollout – you'll have to scroll down to find it).*
- *Select the bone you want to assign the vertices to from the list in the "Parameters" rollout (bearing in mind the rule about how bones names are stored, above).*
- *Type a weight of 1.0 into the "abs weight" box to assign the vertices to the bone*

You can check vertex assignments by clicking on the bone names in the parameter rollouts; each selection will highlight the vertices attached to the link. If the vertices highlight in a color other than red, they may be assigned to more than one bone and should be checked. Reassign problematic vertices using the procedure above.

Skin supports vertex assignment with envelopes similar to the system used by Physique, however it is not recommended for use with the exporter.

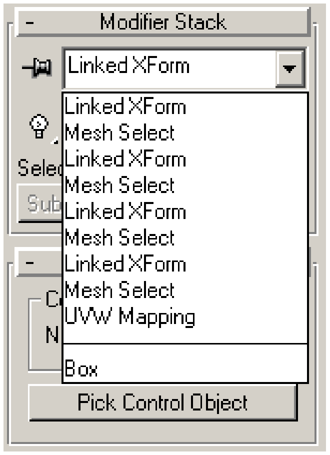
Like *Physique*, *Skin* should be the last modifier in your stack – you can make mapping or material changes after a *Skin* but this is not recommended. There is no good way to preserve vertex assignments in a *Skin* modifier if changes are made to geometry upstream.

The export script that works with the *Skin* modifier will attempt to correct vertices which are assigned to multiple

Using ***Linked Xform***

Max r2.5 and r3 offer another method of attaching vertices to a skeleton. Max's *Linked Xform* modifier takes the selection passed to it by the stack and attaches the selected vertices to a "control object" which works like a Half-life bone. The SDK includes a Max Script exporter script which allows users without Character Studio to make vertex assignments.

Because each *Linked Xform* only handles a single selection - bone relationship, you'll need to create a series of selections followed by *Linked Xform*s to connect your entire mesh. You should use the *Mesh Select* modifier to make your selections. When you've completed your assignments your stack should look something like the image at right.



Modifier stack of a model using Linked Xform *to assign vertices.*

If you select the same vertex in more than one of your *Mesh Select* modifiers, the later selections will override the earlier.

Although it is possible to add *Edit Mesh* or other geometry modifiers after the series of *Mesh Select / Linked Xform* pairs, it is not recommended because it will result in inaccurate previews in Max . For best results try to have the selection / link series of modifiers as the last items in the stack.

Animation Basics

Animation is probably the most complex and time consuming part of creating Half-Life characters. Fortunately the HL exporter supports almost all of 3DStudio Max ' animation tools, so experienced animators should have few problems in adapting their work for Half-Life. If you have not done much character animation before beginning your Half-Life project, it's a very good idea to look at some of the tutorials and guidebooks available in print and on the web. The Character Studio documentation and the 3Dstudio tutorials are good places to start. Useful books include:

Digital Character Animation, by George Maestri (New Riders, 1996)

Character Animation in Depth, by Doug Kelly (Coriolis, 1998)

3D Creature Workshop, by Bill Fleming (Charles River Media, 1998)

There are also several good books available on traditional animation which can be very useful when thinking about how to represent emotion or movement in a character.

Before continuing on you should make sure you are familiar with the following basic animation concepts and Max features:

- *Keyframes*
- *Animation controllers*
- *Walk and run cycles*
- *Interpolation or "tweening"*
- *Inverse Kinematics*

Animation system overview

Every animation in *Half-Life* draws on two sources of data. The basic keyframe data is supplied by the animation SMDs exported from Max . Animation data with no physical component, such as the playback framerate of an animation, or whether a particular animation can be looped, is specified in the QC file. Together the keyframes and other data make up an *animation sequence,* which is created with a `$sequence` command in the QC.

The animation system and the AI only know about sequences called in the QC file. An animation SMD which isn't called in a `$sequence` in the QC file won't be included in the model even if it is properly exported from Max . On the other hand, you can call the same animation several times at different framerates, for example, or with different sound effects and the AI will treat each instance as an independent animations (a simple creature might have a "walk" and a "run" which were used the same keyframes played back at different speeds, for example).

Exporting an animation

Exporting an animation is as simple as choosing the SMD exporter from Max's File > Export menu. The SMD file you create should be saved in the directory indicated by your QC file's `$cd` statement (see *QC file reference*, p. 42). Make sure to save the animation as an "animation .smd" rather than a "reference.smd." Use the SMD export plug-in to export your animation SMDs regardless of which method you used to assign your vertices.

Adding a $sequence command to the QC file

`$sequence` has a number of parameters which are detailed in the QC file reference. Most animations, however use only a handful of `$sequence's` options.

The simplest possible `$sequence` command would look something like this:

`$sequence` ***"sequence name" "smd file"***

> **`$sequence`** is the command which identifies an animation.
>
> ***"Sequence name"*** is a text name for the sequence; when the game reports information back to you – for example, if you are viewing animations in a cycler – this is the name that will be displayed. It's a good idea to enclose this name in quotes.
>
> ***"Smd file"*** is the name of the smd file which supplies the keyframes for this sequence (without the .smd suffix). It's a good idea to enclose this in quotes as well.

Since no other information, this sequence has no special options. It will play back at 15 fps (the default framerate) and will not loop.

Here is `a $sequence` line with more options:

```
$sequence "sequence name"  "smd file" fps 30 LX  loop ACT_WALK 1
```

> The $sequence, sequence name, and smd file parameters are the same as in the first example.
>
> **`Fps 30`** specifies the framerate for this sequence.
>
> **`LX`** means that this animation uses "motion extraction" – meaning the movement portion of the animation will be passed on to the AI or the player movement to be evaluated. LX allows you to set up walk or run cycles which in which the poses repeat but the movement is varied by the AI or player control.
>
> **`Loop`** identifies this as an animation that can be looped seamlessly -- for example, a walk cycle. The pose in a loop animation should be the same in the first and last frame (in point of fact you're saving 1 frame more than a complete loop). Animations with movement extraction turned on don't have to end the loop in the same place – only the same pose.
>
> **`ACT_WALK`** is an action tag, a label which identifies this animation to the AI as a "walk" animation. The uses of the action tag mechanism are described below.

Remember to include a line for every animation in your model. Only animations listed in a $sequence command will be included in your compiled model.

Animating your model

Almost any animation you can create in Max can be exported to a *Half-Life* model. You can use any of Max's animation tools, such as inverse kinematics, noise controllers, expressions, or Max Scripts to generate your animation keyframes. Anything which produces data visible in the Max trackview (either as keyframes or as a controller) can be exported properly. Note that a script which generated an animation on the fly but did not create keyframes or reside in a "script controller" node in the track view would *not* be exported correctly.

Only data relating to the skeleton will be exported, so any animated modifiers or special effects (such as the "flex" modifier) will be ignored by the exporter and will not appear in your *Half-Life* model. As already noted, the exporter will save only rotation and movement data – scales will be ignored.

Time

Unlike Max, the exporter does not care what real-world time unit a frame is supposed to represent. The exporter ignores all of Max's time configuration options except for the animation range (the start and end frames). The speed at which the frames will be played back in *Half-Life* is set by the `fps` parameter in the QC file's `$sequence` command. The frames-per-second setting in the Max file is important only for helping you preview your animations properly.

It is not necessary for animations to start on frame 0 in Max . However animations with negative frame numbers (i.e., frame –10 to 10) can be problematic and should be avoided. You can tell $sequence to use only part of an animation with the **`frames [startframe] [endframe]`** parameter. Thus

```
$sequence "middle" "BaseAnimation"
fps 30 frames 10 20
```

would use only frames 10 through 20 (inclusive) of the animation in the file "Base Animation.smd"

While it is possible to use the `fps` command to stretch or shrink an animation, extreme changes – playing a 30 fps animation back at 2 fps, for example – will often produce artifacts. The artifacts are most pronounced when slowing animations down by large factors. If you need to make an animation much longer (say changing a one second animation to one minute) you might be better off using Max's rescale time ability to create a larger number of keyframes.

Max Keys and SMD file keys

Max supports a number of controller types and many of these include special parameters with their keyframes which control the rate of change between the keys. It is possible to get a variety of subtle effects using such features as ease-in and ease-out values, key curves, and so on. This give animators greater control with fewer keyframes cluttering up the track view.

The exporter will create one keyframe for every frame in the animation – regardless of whether the model's pose at that frame was set as a keyframe or derived from some combination of ease values and so on. If the Max animation and the SMD animation were played back at the same frame rate as each other they would appear identical.

If the Max animation were played back at a much lower framerate, the same parametric controls that ease the movement between keyframes would generate a larger number of in-between poses and the animation's smoothness would be preserved – the relationship between the keys is a basically a mathematical function which can be scaled to cover an arbitrary number of in-between frames.

The animation in the engine, however, is only a series of keyframes with no control over interpolation. If the keyframes are used cover a larger stretch of time, the engine will generate new keys with a simple linear interpolation.

A safe rule of thumb is to generate more keys in Max if you need to slow an animation down to 50 % or more of its original speed.

Interpolation

When your animation is played back in the *Half-Life* engine, the animation system will attempt to display it correctly regardless of fluctuations in the framerate of the game. This means that whether your animation plays back on a machine which can only produce half the framerate for which the animation was intended – or twice that framerate – the animation will be played back so that its duration in real-world time is the same. Faster or slower computers will change the smoothness of the playback, but the integrity of the motion will be preserved as well as possible.

You may experience some interpolation artifacts if there is a very wide variation between the speed at which your animation was created and the final playback speed. If, for example, you created an animation at 10 fps and played it back on a machine capable of 50 fps, you might see some differences between the animation as played in Max and the animation in the engine. Generally animating with a 30 fps framerate will minimize this problem.

Certain kinds of motion are also very vulnerable to frame-rate related strobing. For example a 3-bladed propeller might look good spinning at 30 fps, but may appear to spin backwards at considerably higher or lower framerates. You may find it useful to have such objects spinning at lower speeds, which lessens the likelihood of this occurring.

Basic Animation Tasks

Here are guidelines for some of the more common types of animation.

Movement Animations

Movement animations – typically walking or running but also swimming and flying – are identified to the AI by action tags (see *Animations in the game engine,* P. 35). Animations for player – characters in multiplayer are not controlled by the AI and involve some special issues; see

The AI will use the speed of movement in your animation to navigate the character through the world. To change the character's speed, simply change the animation – no change to the AI code is required. Remember that 1 Max unit equals one inch in the game world when planning your movement animations.

Movement animations will always be loops so they can be repeated as necessary; animators generally refer to them as "run cycles", "walk cycles" and so forth. Unlike some systems, Half-Life does not require you to animate cycles in place – you animate the walk or run with the character moving through space. The motion through space is what tells the system how fast your character will move in the game. By convention "forward" is moving down the Y axis. enabling you to be certain that the character's feet stick properly to the ground.

Movement animations use the `LX` motion extraction parameter as part of their `$sequence` line. `LX` enables the engine to keep track of the model's position and prevents the model from snapping back to its initial position after each play-through of the cycle. If you forget to add the `LX` tag to your sequence, the model will continually pop back to its starting point rather than navigating through the world.

Previewing cycles in Max

Since you are adding an extra frame to the end of your looping animations, your Max animation previews will show a hitch as they loop due to the extra frame. It is a good idea to render your preview at the nominal length of the cycle (i.e. without the extra frame).

If you're having trouble debugging a walk cycle because the character's movement through space makes it harder to see hitches, you can temporarily attach a camera to the root node of the character. Since the camera will be carried along with the moving character you can effectively view the cycle running in place.

Character Studio users can use the In Place Mode button in the Motion command panel to view their cycles in place.

When building cycles, make certain your exported animation **begins and ends with the same pose** (though not the same position in space, naturally). For example, if you want a 1 second walk cycle at 30 fps, you will use 31 frames – 30 frames of animation plus a final repeat of the initial pose. The extra frame provides the animation system the data necessary for proper interpolation -- it will not cause a hitch in the cycle when you play the animation back in the engine.

Action animations

AI controlled behaviors such as attacking, dying, or simply standing around idly are created by simply tagging animations with the Action tag mechanism appropriate to the action. The details of the action – the moment when an attack lands, or when a sound is to be played, even the frame on which a dying character actually dies – are indicated by animation event parameters in the $sequence command. For example the last line in the human grunt's QC file,

```
$sequence    throwgrenade    "grenadethrow"    fps    30    {    event    7    46}
             ACT_RANGE_ATTACK2 1
```

indicates that the grenade is spawned in the game ("event 7") on frame 46. Individual characters have unique events defined in their AI. There are also a series of generic events which any character can access. A list of shared animation events is documented on page 49.

Action animations should not include a meaningful movement component unless you are prepared to alter the AI.

When character's AI finishes one state and moves to another, the system will blend between the last frame of the old animation and the first frame of the new animation over a tenth of a second. You should plan your action animations in ways that anticipate this – you'll want to begin and end your animations in poses that either match exactly or in poses where the tenth of a second blend makes an acceptable transition. AI code can omit the tenth second delay in cases (such as the recoil of a gun) where it is too slow.

Scripted Sequences

Half-life's most powerful storytelling tool is the scripted sequence animation, which allows the animator to create scenes of great complexity without requiring unusual AI or code. Scripted sequences effectively suspend character AI while they are being played, allowing characters to perform actions that would be impossible to them under ordinary circumstances. A simple scripted sequence might involve nothing more than a character moving to a particular location and delivering a speech or activating a button. A complex sequence, on the other hand, could involve several characters interacting – one character carrying another, for example – as well as special effects, sounds, an in-game events such as camera shakes, doors opening, and so on.

Scripted sequences are created in WorldCraft by placing scripted sequence nodes. The scripted sequence node contains controls for specifying:

The monster type or particular character involved in the sequence -- you can set a sequence for any available character of a given type, or only a particular one. You can also set a "search radius" – how far the script will look for a monster to come play the sequence.

An "idle" animation for the target character to play before the sequence is triggered – this will allow you to keep a character in an unusual position until required (the scientist hanging on to a collapsed platform in Half-life's subway station, fore example).

How the character will move to the location of the script -- characters can walk, run, or instantly "pop" to the sequence location when the sequence is triggered, or they can play the sequence from their own locations without moving.

The scripted animation. Any animation can be used in a scripted sequence, even an animation which is also used as an action or motion animation. Scripted sequence animations do not have any special action tags (they are called by name in the WorldCraft scripted sequence node).

When a scripted sequence is playing, all of the character's ordinary behaviors are suspended. This can be problematic if the player manages to interact with the sequence in some unforeseen way (for example, by shooting the character playing the sequence). If you are developing a particularly complex sequence, or one with interlocking animations for several characters, you'll want to control the environment in which the sequence takes place to minimize the possibility of interference.

Scripts can have associated special effects. Scripted sequences can contain animation events that will fire WorldCraft triggers (allowing a scientist to open a door, for example, or causing a func_breakable to explode at an appropriate moment). Sequences can also include audio or effects cues, also included in animation events. Look at the list of events on page 49 for a sense of the kinds of effects you can add to scripts.

If you need to synchronize multiple characters you will have to place a separate scripted sequence node for each character. When you have completed animating your scenes, you will have save a separate Max file and SMD for each character in the scene (having deleted all of the other characters) before exporting the animations. If you attempt to export a scene with multiple characters you'll get a parent mismatch error from StudioMdl. You can fire all of the script nodes with a single trigger, and the animations should play back with proper synchronization and spatial relationships.

Since it is inconvenient to place multiple script node in the exact same sport, you can move the nodes to different locations but keep the animations in the same place with $origin statements.

If you need characters to interact with the world, you can give them props by making the separate "characters" out of the prop models and animating them separately. For example, late in Half-life the player comes to a barricaded door guarded by a scientist with a gun. The gun model is actually a separate monster_generic entity playing back a scripted sequence synchronized to the scientist's hand movements.

You can add simple scripted sequences with "empty" sequences or by calling ordinary AI animations as scripts. You can make one character "chase" another by having the two characters called to run to script nodes, for example. The scripted sequence mechanism is one of the most powerful tools for making your game come alive, and you should think hard about how you can use it to best advantage.

Player animations

Player animations – those controlled by a player rather than the AI – are complicated by a number of factors. Unlike AI animations, there is no way to predict the sequence of actions a player will take; the AI can force a monster to slow down before reversing course, for example, whereas a player character can switch from a full-tilt forward run to a high-speed backpedal instantaneously. Moreover the finesse with which player characters maneuver is highly influenced by the quality of network connections – models controlled by high-ping players will generally take bigger steps (for a single control input) than low ping players' models. For all these reasons the animation system handles multi-player animations in a different manner than AI controlled movement animations.

Gaits

Players, like other characters, can walk or run. The system will interactively adjust the speed of the animation playback to allow for variations in speed, so a character animated to run at Half-life's default speed of 320 units per second will use the run animation, but a character moving at 240 units will play the same animation at ¾ speed. Below a certain threshold (set in code) the system will switch to a walking gait (i.e., a new animation) and play that back at a similarly modulated rate.

Upper body motion

Current half-life multiplayer code uses controllers (see *Controllers*, p. 41) on the vertical rotation of the player model's spine to let the player look left and right while moving. When animating for multiplayer you should keep the player's pelvis steady so that the rotated spine will remain relatively upright. Be sure to test your multiplayer animations with this in mind.

The positions of the player's arms and head are independent of the original run/walk animations as well. You create a single run and walk animation and then a series of single frame poses or short animations for the different weapon holds, rather than requiring a complete set of walk, run, etc. animations for each weapon. It also allows firing animations to have a cyclic length independent of the length of the walk and run animations.

The weapon poses come in two versions, a "ready" or "aim" pose and a short "shooting" animation. Each weapon pose is set up using a blend (see *Animation blending*, p. 50) that controls the vertical orientation of the player's upper body (the horizontal rotation is controlled by the spine controllers, as noted). You'll probably need to create two versions of each weapon pose – one for the standing/walking/running posture and another for crouching. The shoot animations use event tags to indicate at which frame the muzzle flashes are to be played. The muzzle flashes are attached to the player's p_weapon models as described in *Attachment points* for items, p. 38

Player animations are identified to the multiplayer server by index, rather than by name or action tags. The server will use the animation index of the default player model to determine which sequence and frame to play. If a multiplayer model has animations in a different order than the default model, that model will display inappropriate animations (see *Animations in the game engine*, below)

Animation system notes.

Animation length

There is a finite limit of 400 frames in a single animation sequence.

Loops

If you plan on making a looping animation, such as a run or walk cycle, you should be sure your exported animation begins and ends with the same pose (if you want a 1 second walk cycle at 30 fps, you will use 31 frames). The extra frame provides the system the data necessary for proper interpolation; however it will not cause a hitch in the cycle when you play the animation back in the engine. This can be slightly confusing in Max because there will be one extra frame in your Max previews – it is a good idea to render your preview at the nominal length of the cycle and then reset the animation range to include the extra frame before exporting to the engine.

IK target objects

If you are using Max Bones systems to animate your character, you may want to use dummies or other objects as targets for your IK chains. Remember that any target objects must also be present in your reference file, or you will not be able to compile your model correctly. Consider including any target objects in your skeleton by linking them to its root; if you need the target objects to move as if they were unconnected, you can turn off their transformation inheritance (see p. 25)

Animations in the game engine

Once your animations have been exported and called in the QC file, your model is ready to be compiled. But how does the engine know which animations to play, and when?

The engine has two different ways of identifying the animations in a model. Which system is used depends on which function the model is performing. If the animation is going to be called by an AI –

for example in the case of a monster's movement or fighting animations – it will be identified with an action tag. Other parts of the engine, such as the multiplayer code, will identify the animation by its place in the list of animations in the QC file. Finally map entities such as scripted sequences call animations using the "sequence name" specified in the `$sequence` command.

AI animations

"Monster" models – i.e. any model controlled by an AI – use the action tag mechanism. Action tags are added to a `$sequence` line in the QC file to tell the AI that a particular animation sequence corresponds to a particular AI function – for example walking, shooting, or dying. The AI is only aware of animations marked with action tags. Note that the action tag is not the same as the sequence name – a sequence called "walk" that does not have an `ACT_WALK` tag will not be recognized as a walk animation by the AI.

The AI will automatically use the speed and duration of tagged animations in its decision-making process. Thus to change a monster's movement speed, it is necessary only to change its `ACT_WALK` animation – you could even simply change the framerate of the animation in the QC file. The AI will navigate the character using the new speed without code changes.

It is possible for more than one sequence to have a particular action tag. A character with several different idling animations might have a number of animations with `ACT_IDLE` tags, for example. If there are multiple sequences with the same action tag, the AI will choose randomly between the eligible sequences. You can weight that choice by adding a number to the action tag; for example if you have two sequences:

```
$sequence "normal_walk" "normalwalk" fps 30 loop LX ACT_WALK 1

$sequence "silly_walk" "sillywalk"  fps 30 loop LX ACT_WALK 2
```

the "silly walk" animation is twice as likely to be called as the "normal walk". The probability of choosing the any particular animation is its action tag number divided by the_sum of all the numbers for sequences with this tag.

Be sure your action tags have weighting numbers – action tags without selection numbers can cause StudioMdl to crash.

The ability to choose between animations randomly is an enormously powerful tool – it lets animators add many shades of nuance to a particular AI behavior without any changes whatsoever to the code.

Code-controlled animations

Animations not controlled by the AI system do not use action tags to identify sequences. Instead they are identified internally by their position in the list of animations in the QC file. The first `$sequence` in the QC file would be animation 1, the second animation 2, and so on. There is no explicit index or table – the numbers are simply a function of the order in which `$sequence` commands are called in the QC.

This has very important consequences. The code which calls the animations knows nothing about them except their place in the QC file – that is, it simply says "when X occurs play animation 5". An

Multiplayer animations

When the server in a multiplayer game updates the clients, it sends its animation data to them as sequence indexes. The server does not discriminate between different player models. Therefore every player model must have the same index of animations as every other. For this reason the *Half-Life* and TFC multiplayer models call their animations from a shared QC file to prevent the possibility of a mismatch in animation indices. However it is possible for the actual keyframe data in different models to be different, as long as their order is the same and the animations have the same number of frames and frame speed.

accidental reordering of the list in the QC may cause the code to call the wrong animations (indeed, adding one line to the beginning of the `$sequence` list would change the index of every animation in the model and cause all of them to be called incorrectly).

Weapon view models are one example of an entity that calls animations by their sequence index; player models in multiplayer games are another. It is important in working with these types of animations that animators and programmers work closely on these types of animations to avoid confusion.

Calling animations by name

The name which each `$sequence` command includes is the way in which entities in game maps – primarily scripted sequences – can call animations. A scripted sequence node in WorldCraft includes a field in which you can specify animations for the script's active and idle states. The script will activate the appropriate animation when triggered. It's a good idea to let your level designers have an up-to-date copy of the QC files for your models, so that they can cut-and-paste sequence names. There is no error detection when typing sequence names into WorldCraft. For more information on scripted sequences, see p. 33).

Advanced Modeling and animation

Advanced Modeling

Viewmodels

Weapon viewmodels present a couple of unique challenges. They are generally lower in polygon count and texture allowance than characters, but they are always visible to the player at a very close range. To get the best results it is important to know a few tricks:

Setting up View models

To preview your model, set a 90 degree F.O.V. perspective camera at origin (0,0,0) facing along the negative Y axis. This simulates pretty well the player's camera view in the game. Build your model using the camera view as a guide. There are subtle differences, however, between the cameras in the Half-life software, GL, and D3D renderers, so it is not a good idea to assume you will get absolutely identical results in the game. If you are optimizing your model by removing unseen geometry, be sure to leave a little leeway around the bottom and sides of your screen.

You can use a piece of geometry or a Max Tape Measure object to show where your weapon is really pointing. A guideline object about 15-20 feet (180-240 units) gives a good idea of how your weapon is aimed; if the end of the guideline is on the vertical centerline of the screen, about 33% of the way up from the bottom your weapon will appear to be pointing straight ahead in the game.

When a player jumps or falls the viewmodel will lag behind his motion by several frames. In the case of violent accelerations (explosions, etc.) you may see parts of your viewmodel which you thought safely off-camera hanging exposed in mid-air. Be sure to test your viewmodel with jumps, falls, and explosions.

Perspective correction

Because the 90 degree camera view is very foreshortened, if you build your weapon to scale it will look strangely stunted (and not very intimidating) in the camera view. You can apply a forced perspective correction on the model using a Max *FFD* or *Taper* modifier. You can check your results for visual effect in the 90 degree camera view.

Unfortunately if you are modeling hands to accompany your viewmodel, it is difficult to correct their perspective because changing hand and finger positions can easily break the illusion. If you can keep the animation of fingers and wrists to a minimum you can create a more convincing illusion.

Optimization

Since you have complete control over the viewers relationship to the viewmodel, you can greatly optimize your model's polygon usage. Any face on the model which is not visible from the perspective camera can be safely deleted.

You should perform this optimization after animating the model, so that you can be sure which polygons always face away from the camera. You can test an un-optimized version for jump-induced viewpoint shifts (see above) as well. Since deletions are the only safe mesh-editing activity you can perform after attaching a mesh to a skeleton, this is an exception to the general rule about finishing your mesh before attaching it to the bones and animating it.

Clipping problems

There is a problem in the software renderer when the polygons in the viewmodel are clipped through the rear clipping plane of the game camera. This usually manifests itself as black triangles or "holes" in the affected parts of the model.

You can avoid this problem in part by not letting your model's geometry intersect with the game camera's clipping plane. If you set the near clipping plane of your Max camera to 4 units, you'll get a fairly clear indication of which polygons would clip through the Half-Life camera's frustum.

You should also consider subdividing polygons which are in danger of frustum clipping. If you have a ring of vertices more or less parallel to the plane of the camera, you'll keep clipping artifacts to a minimum.

Attachment points for items

Many models may need to have sprites, multiplayer weapons, or other entities associated with them in the game. Each model may have up to 4 attachment points defined in its QC file (see p. 43). Attaching sprites or other entities is done in code – there is no way to preview it in Max.

The attachment points are identified with bones in your model's skeleton. The exact point of attachment is specified in units, expressed in the parent bone's local coordinate space. If you are including an object in your skeleton as an attachment point, it is helpful to make sure that in the reference model it's local coordinate system is rotated to match that of the world (you can adjust it using the hierarchy command panel's "Affect Pivot Only" and "Align to World" buttons"). This makes the trial and error process of specifying attachment coordinates much easier to visualize.

The original shipping version of Half-life used player models to hold attachment points for multiplayer weapons, rather than using attachment points on the weapons themselves. This was changed with the release of Team Fortress Classic – now if a multiplayer weapon model includes an attachment point (say for it's muzzle flash) that attachment point will over-write the corresponding attachment point on the player model itself . In this way all player models can use "attachment 0" for muzzle flashes and "attachment 1" for shell ejection without having to share only four attachment points on the player model; each can specify a unique "attachment 0" or "attachment 1" of its own.

Swappable body groups

Models can have multiple body parts that can be swapped independently of each other. A character might have an optional helmet, pieces of armor, and other equipment which could be combined in various ways to make a number of variants all in the same .mdl file. This is the only exception to the rule that a model can have only one mesh object deforming with it at a time – each body group can be an individual mesh object with it's own vertex assignments.

Body groups must use the same skeleton. The easiest and most reliable way to make body groups is to model all of the bodygroups in the same max file until you are ready to export. Then for each bodygroup delete all of the others and export it separately (be sure to keep a copy of the body group reference max file as well). Body groups animate with the base model and cannot be animated separately – the bodygroup will simply be present or not. Only the base model will have attachment points and controllers.

Body groups are an excellent method of creating visual variety and sharing texture memory. If you have some characters which will appear in large numbers, you should consider body groups as a method for individualizing the characters in a very efficient manner.

Half-life's scientists and soldiers are good examples of how body grouping can create more variety with little additional work.

Advanced Texuring

Texture Groups

Texturegroups are a mechanism for including multiple alternate textures in a model, similar in many ways to bodygroups.

Texture groups are created in the QC file (see the $texturegroup command, p. 47). A texturegroup consists of a series of texture maps that can be applied to the same geometry -- i.e. maps painted for the same UV coordinates. Only one map is applied in 3dStudio – the others are indicated in the QC file and swapped onto the model at runtime. The QC file defines the texturegroup by finding all of the faces which use the member of the group as their texture.

The different maps can be swapped by game code to create an animated texture effect, such as the blinking eye of the Houndeye. You could also use texture groups to create damaged or wounded versions of a texture, or cyclical motions like a rolling tank tread.

All of the textures in a texture group will be loaded whenever a model is active (even if not all are visible) so plan for the extra memory required if you intend to use texture groups.

MipMapping

Half-life models do not currently support Mip-mapping, a technique for optimizing the appearance and memory usage of textures at varying distances. Geometry created in WorldCraft, however, does support mip-mapping. It may be difficult to achieve seamless integration between a character and an environment built in WorldCraft because of this disparity, so plan accordingly.

If texture sampling artifacts are particularly annoying, you may consider implementing a crude form of mip-mapping using texture groups.

Camera Maps

Max's "Camera Map" modifier projects UV coordinates onto a model using a perspective camera as rather than a standard UVW Map gizmo. If you render a scene, then apply the a camera map to the objects in the scene, the rendered image would perfectly match the geometry underneath – in essence this is the same as the texture registration method outlined on page 22.

Using the camera Map modifier and UVW Map is a powerful way to optimize texturing objects where you are sure you'll have control over the viewer's eye point – the most obvious example being weapon viewmodels. The advantage of using a camera map is that you'll have a distribution of pixels based on how well the viewer can see the object – close up areas get a lot of pixels, while far off areas or those seen very obliquely get but a few. The major limitation of this technique is that it works best when there is not much change in the relationship of the player's eye point to the model – a weapon viewmodel, for example, assuming the weapon's animations were not very extreme.

Make a set of high res textures for you model (you can even include rendering functions such as specularity and bump maps which Half-Life does not support directly). Render your image with a camera representing the eye point you expect to have when viewing the model -- in the case of a viewmodel, this would be the player's ordinary view of the model, for example. Then apply the rendered image to the model as a camera map and a Half-life compatible bitmap texture.

If you seem to be wasting a lot of space on empty parts of the render, or you need to set the resolution of your texture to a fixed size, first crop the texture and then use *UVW Unwrap* to scale the texture coordinates to match the new size. The resulting mapping is the most efficient possible use of texture pixels for the given eye point and model.

Team colors

Half Life allows you to remap two color ranges for multiplayer models, allowing you to use the same models and textures for different teams in multiplayer. To support team color remapping, textures must be named "DM_Base.bmp" (only one map with this name is allowed per model, naturally). Remappable textures are 8 bit .bmp files like regular textures. There are two remappable ranges of 32 colors each, from palette index 193 through 224 and 225 through 256. The colors in these ranges can be reset in code, or using the standard selector in the multiplayer options section of the launcher. Remapping affects hues only – the saturation and value of the new color are identical to those of the old color. For this reason pure grayscale, with a saturation of zero, would not remap properly.

Chrome Maps

The *Half-Life* engine offers a unique kind of texture that you can use to simulate reflective or shiny surfaces. A *Chrome map* is similar to what many 3-d packages refer to as environment or reflection maps. Rather than gluing the texture to the vertices of the model, a chrome map moves with viewer's eye, giving the impression of a reflection that moves across the surface of a stationary object. Unlike a true reflection map, however, the chrome map is never changes its orientation relative to the viewer – thus you cannot walk around a chrome map object and see the "back" reflection. For this reason your chrome maps should be indistinct and suggestive, rather than mirror-like images intended for clear reflections. Chrome maps are best for somewhat irregular surfaces where their limitations can't be perceived as easily. Flat planes, which the view would expect to have a real mirror-like reflection, will usually break the illusion.

All chrome maps must be 64 x 64 pixels. They can use any palette. To indicate to the exporter that a maps is a chrome map, simply include the word "Chrome" in its name, i.e., "MyChrome.bmp"

When a chrome map is projected onto the model, it is stretched at the top and sides to give a spherical impression. The effect is akin to wrapping a square piece of wrapping paper around an object in a single motion.

Chome maps are wrapped around objects by smoothing group. This means that the bounding volume for the spherical projection is based on the bounding box of all the faces in the model which share this chrome map and have a common smoothing group ID applied to them. To create an impression of greater complexity within a reflective object, you can break up the "chrome" area into a number of smoothing groups, each of which will use a unique projection of the chrome and thus will look somewhat different from the others

Advanced animation

Controllers

Controllers are Half-life’s mechanism for assigning direct control over a bone to the game code. Each controller can set the rotation or position of a bone directly (relative to it’s original location in the reference model) independent of the animation of the overall model. A simple example is the moving mouths on the Half-life scientists; the jaw bones were hooked to a controller and the controller was fed rotations based on the amplitude of the speech .wav files as they played. Controllers are extremely useful for giving game code easy, precise access to character behaviors.

Controllers are defined in the QC file with $controller statements. The statement names a controller channel (from 0 to 7), the affected bone, the axis of control and the limits of the movement. A model can contain up to eight controller channels, with each channel may controlling any number of bones. Each controller handles either the rotation or the translation of a single axis. A bone may have more than one controller if it is moving and / or rotating in more than one axis.

In this example:

```
$controller 0 "radar_left" ZR 45 -45

$controller 0 "radar_right" ZR -45 45

$controller 1 "radar_left" Z 0 10

$controller 1 "radar_right" Z 0 10

$controller 2 "aim_up" XR -10 40
```

the bones “radar_left” and “radar_right” each have two controllers influencing them. The data from channel 0 will cause the two bones to rotate in opposite directions around their local Z axes (notice that the limits on the rotation are in reverse order). The data from channel 1 will cause them to move along their local Z axes. The “aim_up” bone is controlled by channel 2 and will rotate around it’s local X axis.

Translation controls are specified in world units, with 0 being the object’s position in the reference file. Rotation controls are specified in degrees, again with 0 being the rotation in the reference pose.

Animation blending

Animation blending is another mechanism that allows game code to control a model directly. When the soldiers in Half-Life rappel down from the Osprey, they point their guns using a blend between to short animations – one pointing up and another pointing down. By manipulating the amount of blending between the two sequences, you can create a number of poses or animations without using controllers. Blends are good for situations where you need to adjust between two positions involving several bones; controllers are good for situations where a single bone is used.

The QC file syntax for setting up blended animation is included in the description of the $sequence command on page 50.

QC file reference

QC Files

The QC file is essentially a script which tells StudioMDL how to compile a model out of the SMDs and textures you have created.

The minimum set of commands needed to create a model are:

`$modelname`	tells StudioMDL where to place the MDL and what to call it.
`$cd`	tells StudioMDL where to find the source files for the model
`$cdtextures`	tells StudioMdl where to find the textures for the model.
`$body`	Identifies a reference SMD
`$sequence`	creates an animation sequence

Every QC file must have at least one of each of these statements

Formatting

QC files are plain text files and can be created in any text editor. The files are "white space" delimited, meaning that spaces, tabs or returns can be used to separate arguments. For best legibility, keep all commands on their own lines and use curly brackets (see below) to split complex commands over several lines.

You can include comments in your QC files by preceding lines with double slashes "//" or by using C language commenting:

```
/*
<comment>
*/
```

to enclose multi-line comments.

Commands and arguments are case –insensitive. Pathnames, file names, and other text data should be enclosed in quotes

Compiling QC files into Models

In most case you compile a model by invoking StudioMdl and passing it a path to the QC file you wish to compile, e.g.:

```
StudioMdl models/barney/barney.qc
```

will compile the barney.qc file. The QC will tell StudioMdl where to find the source files and where to output the resulting .mdl file.

If you are comfortable with windows batch files, here's a useful trick for streamlining your work with StudioMdl. Create a batch file which will navigate to the place where you want to call StudioMdl (usually this will be just above the top level of your project – for details on how to organize your directory structure for efficient production see *Setting up your work files* on p.9 Then add a call to StudioMdl with a parameter variable (%1). This will pass whatever argument the batch file gets on to StudioMdl. When you have your batch file complete, use the windows file type editor to associate files with the .qc extension with your new batch file. Now you can compile a QC file simply by double-clicking on the QC or dragging it onto your batch file's icon.

Here's a sample batch file which will also pop up a window for you to view the results of the compile operation. You may have to edit the path names If you don't use the standard directory setup described on p. 9.

```
echo off
title compiling QC for %1
color 3F
cd \
cd Sierra\Half-Life
studiomdl %1
pause
```

Advanced Max Script users can also write scripts which call StudioMdl directly using Max's `DOSCOMMAND()` function.

Path names

QC files support both relative and absolute path names. We suggest using relative pathnames (see *Setting up your work files,* p. 9) to facilitate cooperation. Pathnames will be relative to the directory from which StudioMdl is invoked – usually the *Half-Life* root directory.

When specifying SMD file paths, do not include the .smd suffix. Enclose all path names in quotes.

QC command listing

$attachment <ID#> <bone> <X> <Y> <Z>	Identifies an attachment point for sprites, weapons, and shell ejection. There is a limit of 4 attachment points per model <ID#> is unique identifying number from 0 to 3. <bone> is the name of the bone to which the attachment is attached, enclosed in quotes. <X> <Y> <Z> specifies the location of the attachment point in the parent bone's local coordinate system (e.g. 0 0 0 is at the bone's location, 0 12 0 would be 12 units from the bone in the bone's local Y direction). **If an attachment point is specified in a multiplayer weapon model (e.g. p_TFautocannon.mdl) it will override the corresponding attachment point in the parent player model.** This allows you to have different muzzle flashes or ejection points for every weapon model.

$bbox <X> <Y> <Z> <X2> <Y2> <Z2>	Defines the character's bounding box. <X> <Y> <Z> and <X2> <Y2> <Z2> are opposite corners of the bounding box. The bounding box must be centered at the world origin, so X = -X2 and Y = -Y2. Z should be 0 and Z2 will be the height of the bounding box.
$body studio <path> [reverse]	Specifies the smd file to use as a reference. Studio is a tag telling the system to load a reference SMD. <path> is the path to the reference SMD file, enclosed in quotes. **Do not** use the .smd suffix in the path; e.g.: "/models/test/body" not "models/test/body.smd" [reverse] is an optional tag which will flip the normals on the entire mesh if Max exports them facing the wrong way. If your model appears turned inside out in the engine, add reverse to the $body line.
$bodygroup <name> { studio "partreference" studio "part2reference" blank … }	Defines bodygroup named <name>. Body groups are models that be swapped by code to provide variations on a character, e.g.. The scientist's heads in *Half-Life.* All of a model's bodygroups should be attached to copies of the same skeleton (this is the only way in which a skeleton can have more than one deformable mesh attached to it). Bodygroups will move and deform with the character as it animates. A "blank" bodygroup is a dummy entry which will not be visible in the game. It is used as a standin for objects, such as weapons, that may not be present in all instances of a character. The list of parts in the $bodygroup command is enclosed in curly brackets. Each entry is either the tag **studio** followed by a reference SMD file name enclosed in quotes, or the tag blank indicating an empty bodygroup. Do not include the .smd suffix in filenames

$cbox <X> <Y> <Z> <X> <Y> <Z>	
$cd <path>	Specifies the path to the source files for the model.
$cdtexture <path> [<path> ...]	Specifies the path to the location of the model's 8 bit textures. You can specify multiple paths (separated by spaces) if you have textures stored in more than one place. If you have multiple texture paths all file names must be unique.
$cliptotextures	
$controller <Id#> <bone> <axis> <limit1> <limit2>	Gives control of one axis of a bone's rotation to the AI. Each axis of rotation needs its own controller. There is a limit of 4 controllers per model. <ID#> is a unique number from 0 to 3 identifying the controller. <bone> is the name of the bone to be controlled. It should be enclosed in quotes. <axis> is the local axis of the bone to be controlled <limit1> and <limit2> are the extreme values that the controller can rotate to (in degrees). You can reverse the motion of the controller by reversing the order in which the limits are specified.
$externaltextures	Optional command, specifies that the textures for the model will be stored separately, so information in the model can be accessed without loading the models textures.
$eyeposition <X> <Y> <Z>	Sets the position (in worldspace) of the point from which the AI will make the character's vision checks.
$flags <#>	
$gamma <src value>	Optional parameter specifies a gamma value for the textures in the completed mdl. Defaults to 1.8
$hbox <group #> <bone> <X> <Y> <Z> <X2> <Y2>	Specifies the hit detection box for the

<Z2>	specified bone. <group#> is the id number of the body part (head, leg, arm, etc) to which this bone belongs <X><Y><Z> and <X2> <Y2> <Z2> are opposite corners of the hit box, defined in local bone space. Characters with no defined $hboxes will have a best-fit set of hit boxes generated automatically. All of the automatically generated boxes will be assigned to group 0
$hgroup <group #> <bone>	
$include <QC file path>	Inserts the contents of another QC file at the current point in the QC. Contents are treated as if they were typed into one file. Useful for organizing characters with large numbers of sequences, or for sharing animations between multiplayer models to ensure identical sequence indices <QC file path> is the path to the other qc file, enclosed in quotes.
$mirrorbone <bone>	
$modelname <path>	Specifies the name of the completed mdl file. The filename should include the .mdl suffix
$origin <X> <Y> <Z>	Offsets the entire Max file so this point is now at 0,0,0. Equivalent to moving the whole scene by –X ,–Y, –Z units. **Affect all subsequent sequences until a new $origin is encountered**. If you intend to globally reposition a model and its animations, be sure to place the $origin command before the $body / $bodygroup statements.
$Rotate < # degrees>	Rotates the animation sequence around the Z (vertical) axis by the specified number of degrees. Rotate 90 would turn an animation facing forward into one facing left, for example. **Affects all subsequent statements until another $rotate is encountered.** If you intend to globally rotate a model and its animations, be sure to place the $rotate

	command before the $body / $bodygroup statements.
$scale <factor>	Globally scale the model by this factor. You can omit this command unless you need to rescale a model from its default size. **Affects all subsequent statements until another $scale is encountered**. If you intend to globally scale a model and its animations, be sure to place the $scale command before the $body / $bodygroup statements.

$sequence <name> <path> {
[<blend smd path> ...]
[fps <#>]
[loop]
[frame <start> <end>]
[origin <X> <Y> <Z>]
[rotate <angle>]
[scale <#>]
[blend <axis> <start> <end>]
[LX]
[LY]
[LZ]
[event <#> <frame> [options]]
}
	See The $Sequence Command, below.

$sequencegroup <name>	
$sequencegroupsize <# in KB>	Sets the size for the

$texturegroup <name>
{
{“basetexture” “basetextureB” “basetextureC”}
{“texture2” “texture2B” “texture2C”}
{“texture3” “texture3B” “texture3C”}
…
}
	Creates a texturegroup using the faces which have the texture(s) <basetexture> etc. applied to them in the reference SMD file.
	
	Each group is enclosed in brackets. A group may contain one or more textures; when the texturegroup is swapped, each texture will be swapped for the corresponding entry in the new group, i.e. “basetexture” will become “texture2”, “basetextureB” will become “texture2B” and so on. Texture names are enclosed in quotes.
	
	All of the groups should be enclosed in one pair of brackets as in the example.

The $sequence command

The $sequence command has a large number of optional parameters. The minimum required parameters for a $sequence are the identifying name for the sequence (enclosed in quotes) and the path to an SMD file (enclosed in quotes, with no .smd suffix).

If you have only a couple of parameters to add to the basic name-SMD pair, you can just include them on the command line, e.g.:

```
$sequence “sample” “sample_animation_file” fps 15 loop
```

If your $sequence has a large number of optional parameters, you may find it useful to enclose the arguments of the sequence in curly brackets {} so you can format the parameters more legibly. The system will see:

```
$sequence “confusing” “models/test/samplesequence” fps 30 LX loop origin
0 -10 -10 rotate 90
```

and

```
$sequence “confusing” “models/test/samplesequence”
       {
       fps 30
       lx
       loop
       origin 0 -10 -10
       rotate 90
       }
```

as being the same, however the second example is much more readable.

Animation events should be enclosed in their own set of brackets. Bracketed animation events can reside inside a bracketed set of $sequence parameters, e.g.:

```
$sequence “confusing” “models/test/samplesequence”
       {
       fps 30
       lx
       loop
       {event 6 10 }
       {event 1008 20 “funkysound.wav”}
       }
```

For more details about using animation events, see

Animation events, below.

Sequence playback parameters

Fps <#>	Sets the target framerate of the animation in frames per second. If no fps value is supplied the animation will default to 30 fps. Systems unable to match the target framerate will still preserve the real-time length of the animation (see *Interpolation*, p.31)
LOOP	A tag that identifies an animation intended to loop seamlessly. For details on creating looping animations see *Loops,* p. 35.

`Frame <start#> <end#>`	Tells the sequence to use only a subset of the frames in the animation SMD. Sequence will use frames <start #> through <end #>, inclusive and ignore others. Since animations can start at an arbitrary time , Start# and end# are absolute (i.e., frame 21 is time 21 in the original Max file – not the 21st frame of the SMD file).

Moving, scaling and rotating sequences

If these parameters are properly enclosed in a parameter block, they will affect only this sequence (unlike the command versions, $origin, $rotate, and $scale which affect all subsequent QC statements).

`Origin <X> <Y> <Z>`	Offsets the entire animation so point <X>,<Y>,<Z> is now at 0,0,0. Equivalent to moving the whole scene by –X ,–Y, –Z units.
`rotate <# degrees>`	Rotates the animation sequence around the Z (vertical) axis by the specified number of degrees. Rotate 90 would turn an animation facing forward into one facing left, for example.
`Scale <# factor>`	Scales the animation (starting from origin) globally, by <# factor>. Useful primarily for adjusting animations for a character that has been scaled with the $scale command.

Animation events

Animation events are flags in the animation sequence representing interactions between the character and the world. Animation events tell the AI at what moment an attack is launched, or a character dies, when a scripted event (such as a scientist opening a door) has happened, and so on.

A sequence may have multiple animation events (for example a walk animation might have a footfall sound for each foot hitting the ground. Some scripted sequences can have quite a few events.

Events are enclosed in curly brackets. The simplest event calls look like **`{event <#> <frame>}`** where **`<#>`** is the number which tells the system what kind of event has occurred and **`<frame>`** tells the system at which frame the event takes place.

Events which call sound files have an additional parameter, which is the path to the sound file or sentence file to play (relative to the sounds directory). Events which trigger map entities have an extra parameter for the name of the trigger (enclosed in quotes).

Events with ID numbers below 1000 are specific to individual monsters, so the scientists' event 1 is different from the Vortigaunts' event 1. The meaning of monster specific events is indicated in the monster AI code.

Events with numbers in the 1000 range are scripted sequence events. Events with numbers in the 2000's are for monsters (only). Events in the 5000 range are client-side only, and are used primarily for muzzleflash or sound effects for weapon viewmodels.

	Scripted Sequence events	**Extra parameters**
1000	Character dead at this point	
1001	Sequence un-interruptible from this point	

1002	Sequence interruptible from this point	
1003	Fires a trigger in the map	Trigger name
1004	Play .wav file	.Wav file path
1005	Play sentence file	Sentence file path
1006	Do not send character back to floor at end of script	
1007	Go to this animation after script completes	Animation sequence name
1008	Play named .wav file through voice channel	.Wav file path
1009	Play random sentence group (25 % chance)	
1010	Character is alive at this point	
	Monster specific events	
2001	Monster drops light body	
2002	Monster drops heavy body	
2010	Monster plays swing or swish sound	
2020	Monster has turned 180	
	Clientside events for viewmodels	
5001	Muzzleflash on attachment 0	Muzzle flash sprite path
5002	Spark on attachment	
5004	Emit a sound	Wav file path
5011	Muzzleflash on attachment 1	Muzzle flash sprite path
5021	Muzzleflash on attachment 2	Muzzle flash sprite path
5031	Muzzleflash on attachment 3	Muzzle flash sprite path
6001	Eject a brass shell from attachment	

Animation blending

To set up a blending animation, include a second SMD file path after the first in your $sequence line. Then follow it with the `blend` tag, which allows you to specify the limits for the blend in the same manner as for a $controller. So a simple blend statement would look like this:

```
$sequence "blend" "up_pose" "down_pose" blend XR 45 -45
```

The two animation SMDs in a blend $sequence have to have the same number of frames.

Motion extraction

Movement animation sequences should include one or more motion extraction tags. Motion extraction allows you to animate walks and runs normally, rather than having to create cycles in place (see p.32).

The Motion extraction tags are LX, LY, and LZ. They enable motion extraction for the X Y and Z axes respectively. However, the engine's internal coordinate system is rotated 90 degrees around the vertical axis from Max's system. The LX parameter therefore extracts motion in the Max Y dimension, and LY extracts Max's X motion.
This is the only place where you should have to worry about the difference in coordinate systems.

Template QC file

This is an example of a generic QC file. You can use it as a reference when building your own QCs.

```
// Sample QC file
// Comments go here
// Created April 7, 2000 by SJT
$modelname Mymod/models/modelnamegoeshere.mdl

$cd models/samplemodel
 // the source is probably in Half-Life/models/samplemodel.

$cdtextures “models/samplemodel/8 bit textures”
 // if pathname includes spaces, enclose it in quotes

$scale 1
 // scale the entire thing by this scale – in this case, don’t change it.
$body studio “TheReferenceSMDFile”
// add “reverse” if the normals are flipped

$sequence “idle” “IdleAnimationSMDfile” fps 30 loop ACT_IDLE 1
$sequence “walk” “WalkAnimationSMDfile” fps 30 LX loop ACT_WALK 3
$sequence “AlternateWalk” “OtherWalkAnimationSMDfile”
            {
            fps 15
            LX
            Loop
            ACT_WALK 1
            {event 1 15 “samplesounds/Samplesound.wav”}
            )
// this is a way of formatting a line with a lot of options.
```

SMD file format

The SMD file format is a simple text file, easily edited in any text editor. SMD files are carriage return sensitive – each line must end with a carriage return. the file should also include at least one carriage return after the final end statement. White space is the only delimiter; any combination of tabs and spaces can be used to separate values (for this reason multiword names should be enclosed in quotation marks (").

SMD files come in two version, Reference and Animation files. The two are identical except that animation SMDs omit the triangle and texture map data in reference SMDs and contain bone position/rotation data for every frame.

Here are the sections of an SMD file, considered in order:

Header data:

The only header data required is the tag “version 1” in the first line of the file

Node tree data:

Builds a list of all the bones in the skeleton. Each bone has a unique ID number, a unique text name and a pointer to the ID number of its parent. Children of the world use –1 as their parent ID. ID numbers are integers (starting from 0). Names are text, enclosed in quotes. Names may contain spaces.

`nodes`

Starts the node tree data block

`<ID> "Bone Name" <parent ID>`

`<ID>` is a the ID number for this bone. **`"Bone Name"`** is a text name for the bone, enclosed in quotes. **`<Parent ID>`** is the ID number of the bone's parent. Children of the world (unparented objects) have a parent ID of -1

```
. . .
. . .

. . .
```

`end`

Ends the node tree data block

Skeleton Pose data

This block contains the position and rotation data for every bone in the skeleton. In an animation SMD there will be a "time" block for every frame in the animation. In reference SMDs there will only be one time block.

The skeleton block is begun by the "skeleton" tag and ended by an "end" tag. Time block begin with "time <frame>" and end when a new "time" tag is encountered.

`Skeleton`

Begins the skeleton pose block

`Time 0`

Begins this time block

`<ID> <PosX> <PosY> <PosZ> <RotX> <RotY> <RotZ>`

`<ID>` is the bone **`<PosX>, <PosY>`** and **`<PosZ>`** are the position in world units (good to 6 significant digits). **`<RotX> <RotY>`** and **`<RotZ>`** are local Euler rotations in radians. Bones which are not children of the world report their position and rotations in their **parent's local space**.

```
. . . . . . .
. . . . . . .
. . . . . . .
```

`Time 1`

Begins next time block. Every bone has a pose entry for every frame. Reference SMD's export only one frame.

```
. . . . . . .
. . . . . . .
. . . . . . .
```

`end`

Ends skeleton pose block

Triangle block

The triangle block contains a list of triangles. Each triangle is preceded by a bitmap texture file name (filename only – path data is supplied in the QC file). Each vertex of the triangle then reports its parent bone's ID, the vertex's position, the vertex's texture coordinates, and its normals.

The Triangle block is begun with the "triangles" tag and ends with the "end" tag.

`Triangles`

Begins the skeleton pose block

`"bitmapname.bmp"`

Name of the bitmap file for the texture assigned to this face. Includes .bmp suffix. Enclose in quotes.

`<Parent> <PosX> <PosY> <PosZ> <NormX> <NormY> <NormZ> <TexU> <TexV>`

Parent is the ID number of the vertex's parent bone. **`> <PosX>, <PosY>`** and **`<PosZ>`** are the position of the vertex in world space. **`<NormX>, <NormY>`** and **`<NormZ>`** are the components of the vertex's normal vector (normals may be interpolated if triangle is part of one or more smoothing groups). **`<TexU>`** and **`<TexV>`**are the texture coordinates for this vertex.

```
. . . . . . . . . .

. . . . . . . . . .
```

Each triangle contains 3 vertex records

`"nextBitmapName.bmp"`

new bitmap file begins a new triangle listing

```
. . . . . . . . . .
. . . . . . . . . .
. . . . . . . . . .
```

`end`

ends triangle block.

Sample SMD file

This sample is a reference SMD. Animation SMD's would have multiple time blocks and no triangles block.

`version 1`	a tagline. required by all smds
`nodes`	begins the skeleton parent list
`0 "bone01" -1`	this bone has the ID #0, it's called "bone01" and its parent is ID -1 (i.e. the world)
`1 "Bone02" 0`	this bone is ID #1, it's called "Bone02" and its parent is bone 0 (i.e., "Bone01")
. . .	etc., etc.
`end`	ends the skeleton parent list
`skeleton`	starts the list of bone positions and rotations
`time 0`	Begins a time block for time 0. In an animation SMD you'll have multiple

	blocks. A reference SMD has only one.
0 -0.31545 0.00000 0.94637 0.00000 0.00000 0.00000	Bone ID 0 is at -3.1545, 0,.94637 and rotated 0,0,0 (Euler angles IN RADIANS - not degrees!)
1 8.83281 0.00000 29.0221 0.00000 0.31562 0.00000	Bones with a parent other than -1 report their position and rotation in their parent's local space. So bone ID 1 is 8.8 units from bone ID 0 in Bone ID 0's local X direction, and 29.0221 units in Bone ID 0's local Z direction. (note that in this case, since bone ID 0 is rotated to 0,0,0 it the local and world locations will be the same) Rotations are likewise relative. This bone is rotated .31562 from the parent's Y axis (about 18 degrees).
. . .	etc., etc.
end	the end of the skeleton position/rotation section
triangles	starts the triangle list
"sample.bmp"	the texture name for this face (texture location is specified in the QC file)
3 -21.95 8.83 0.07 0.0 .707 0.0707 0.5122 0.9109	the first vertex of this triangle is attached to bone ID 3. It's position IN WORLD SPACE is (-21.95, 8.83, 0.07). The vertex normal vector is (0.0, .707, .707) - this value is calculated taking account of smoothing groups. The texture coordinates of this vertex are .5122 in the U texture direction and .9109 in the V texture direction.
3 -21.1 8.5 1.00 0.0 0.50 0.50 0.61 0.932	the second vert of the triangle is also attached to bone ID 3: It's at (-21.1, 8.5, 1.0) and it's normal is (0,.5,.5) . Tex Coords U:.61 V:.932
2 -22 9 1.1 0.0 1.0 0.0 0.66 0.944	the third vert of the triangle is attached to bone ID 2. The position is (-22,9,1.1), the normal is (0,1,0) i.e., straight forward and the texture coords are U:.66 V:.944
"sample.bmp"	a new face record starts with a new texture listing, even if it has the same as the texture in the previous face.
. . .	etc., etc.
end <CR>	end of the triangle list and of the file

StudioMDL reference

StudioMdl.exe (by default found in Half-Life/Valve/Bin) is the program which compiles half-life .mdl models from SMDs, bitmap textures, and QC files. It is a DOS command line program. Ordinarily StudioMdl is run with only the name of a QC file as an argument. The following options are available for special purposes:.

The command line options of interest are:

-a <value>	Blends surface normals together into single surfaces, optimizing the model in memory. The default value is 2, which should suffice for everything you do. It can be played with at your discretion, however.
-h	Output hitbox info. If you run with this option and redirect your screen output into a text file, StudioMdl will generate a list of the hitbox sizes for each bone of your creature. It generates these hitboxes automatically, You can cut and paste that information back into the .qc file and edit the information if necessary. Doing this will allow for a more accurate representation of the model for determining whether or not shots hit the creature.
-i	Ignore warnings. Useful if there are undone aspects to the model, yet you wish to test it out anyway.
-t <bmpname>	Replace all textures on the model with a single texture. Useful for testing smoothing groups, but can also be mimicked with the "r_drawentities" console variable.
-t <sourcetexture> <replacetexture>	This allows you to replace one texture within a model with another texture.

Command line options

Reading output

Action Tags

This is a list of the action tags recognized by the AI system. For more details about how action tags interact with the AI code, see the relevant documentation in the SDK.

Remember, when calling action tags in the QC file you must specify a weighting number (use 1 if you don't need the weighting functionality). Action tag weights are described in detail on page 36.

No sequence can use more than one action tag. If you want to use the same animation for more than one action, call the SMD from multiple $sequences, with a different action tag for each.

Many actions will need animation events to make them fully functional, i.e.: to indicate when an attack connects, when a hit lands, or when a sound should play. Monster specific animation events are commented in the AI code. Generic animations events are documented on p. 49.

Action Tag	*Description*
ACT_ARM	Activate weapon (e.g. draw gun)
ACT_BARNACLE_CHEW	Barnacle is holding the monster in its mouth (loop)
ACT_BARNACLE_CHOMP	Barnacle latches on to the monster
ACT_BARNACLE_HIT	Barnacle tongue hits a monster
ACT_BARNACLE_PULL	Barnacle is lifting the monster (loop)
ACT_BIG_FLINCH	Large reaction to non-specific hit
ACT_BITE	This plays one time eat loops for large monsters which can eat small things in one bite
ACT_COMBAT_IDLE.	Agitated idle, played when monster expects to fight
ACT_COWER	Display a fear behavior
ACT_CROUCH	The act of crouching down from a standing position
ACT_CROUCHIDLE	Hold body in crouched position (loop)
ACT_DETECT_SCENT	This means the monster smells a scent carried by the air
ACT_DIE_BACKSHOT	Die hit in back
ACT_DIE_CHESTSHOT	Die hit in chest
ACT_DIE_GUTSHOT	Die hit in gut
ACT_DIE_HEADSHOT	Die hit in head.
ACT_DIEBACKWARD	Die falling backwards
ACT_DIEFORWARD	Die falling forwards
ACT_DIESIMPLE	Death animation
ACT_DIEVIOLENT	Exaggerated death animation
ACT_DISARM	Put away weapon (e.g. re-holster gun)
ACT_EAT	Monster chewing on a large food item (loop)
ACT_EXCITED	For some reason monster is excited. Sees something he really likes to eat or whatever.

ACT_FALL	Looping animation for falling monster
ACT_FEAR_DISPLAY	Monster just saw something that it is afraid of
ACT_FLINCH_CHEST	Flinch from chest hit
ACT_FLINCH_HEAD	Flinch from head hit
ACT_FLINCH_LEFTARM	Flinch from left arm hit
ACT_FLINCH_LEFTLEG	Flinch from left leg hit
ACT_FLINCH_RIGHTARM	Flinch from right arm hit
ACT_FLINCH_RIGHTLEG	Flinch from right leg hit
ACT_FLINCH_STOMACH	Flinch from stomach hit
ACT_FLY	Fly (lx loop)
ACT_FLY_LEFT	Turn left in flight
ACT_FLY_RIGHT	Turn right in flight
ACT_GLIDE	Fly without wing movement (lx loop)
ACT_GUARD	Defend an area
ACT_HOP	Vertical jump
ACT_HOVER	Idle while in flight (loop)
ACT_IDLE	Default behavior when nothing else is going on (loop)
ACT_IDLE_ANGRY	Alternate idle animation in which the monster is clearly agitated. (loop)
ACT_INSPECT_FLOOR	For active idles -- look at something on or near the floor
ACT_INSPECT_WALL	For active idles -- look at something directly ahead of you (doesn't have to be a wall or on a wall)
ACT_LAND	End of a jump
ACT_LEAP	Long forward jump
ACT_MELEE_ATTACK1	Attack at close range
ACT_MELEE_ATTACK2	Alternate close range attack
ACT_RANGE_ATTACK1	Attack with ranged weapon
ACT_RANGE_ATTACK2	Alternate ranged attack
ACT_RELOAD	Reload weapon
ACT_ROLL_LEFT	Tuck and roll left
ACT_ROLL_RIGHT	Tuck and roll right
ACT_RUN	Run (loop)
ACT_RUN_HURT	Limp (loop)
ACT_RUN_SCARED	Run displaying fear (loop)
ACT_SIGNAL1	Signal
ACT_SIGNAL2	Alternate signal
ACT_SIGNAL3	Alternate signal
ACT_SLEEP	Sleep (loop)

ACT_SMALL_FLINCH	Small reaction to non-specific hit
ACT_SNIFF	This is the act of actually sniffing an item in front of the monster
ACT_SPECIAL_ATTACK1	Monster specific special attack
ACT_SPECIAL_ATTACK2	Monster specific special attack
ACT_STAND	The act of standing from a crouched position
ACT_STRAFE_LEFT	Sidestep left while maintaining facing (loop)
ACT_STRAFE_RIGHT	Sidestep right while maintaining facing (loop)
ACT_SWIM	Swim (loop)
ACT_THREAT_DISPLAY	Without attacking monster demonstrates that it is angry
ACT_TURN_LEFT	Turn in place quickly left
ACT_TURN_RIGHT	Turn in place quickly right
ACT_TWITCH	Twitch
ACT_USE	Use item
ACT_VICTORY_DANCE.	Victory display after killing player
ACT_WALK	Walk (loop)
ACT_WALK_HURT	Limp or wounded walk (loop)
ACT_WALK_SCARED	Walk with fear display (loop)

Working with *Half-Life* Content from the *Half-Life* SDK

The Half-Life SDK includes Max models, textures, QC files and other resources from the Half-Life. The models included are generally stored in Max r1.2 format. If you are using Max r3 or later, you may find problems working with Half-Life models, due to file format incompatibilities between Max versions. We have observed irregular but fairly common problems with opening Max r1.2 files in Max r3. Generally opening the Max files in Max r2 or r2.5 and resaving them will minimize problems.

Many of the original models were created using the Texturizer plug-in from Sven Technologies. Texturizer performs many of the same functions as *UVW Unwrap.* If you need to work with content that was built with the Texturizer plug-in, you can order a copy of the *SurfaceSuite Pro* package from Sven's website at www.sven-tech.com (phone 415-625-0310). If you are only interested in working with animation or QC data you do not need Texturizer – textures may appear garbled in your Max windows but the model will compile correctly if you use the original reference SMD files.

If you are working with Max 1.2 you will need the original version of the SMD exporter, which contains the files SMDexp.dle and Smdexp.ilk. Place the files in your max Plug-ins folder. This version of the exporter requires that you export a .vph file from within your Physique modifier as well as the reference SMD. The vph file helps StudioMdl assign vertices to bones in the compiling processes. To save a .vph file use the "file" button when the command panel is open to your *Physique* modifier. When working with Half-life content, be sure you have the .vph files as well as the SMD's and textures before attempting to recompile the models.

The SDK also includes a version of the exporter recompiled for Max r2. It functions identically to the original export plug-in.

In Max 3 and later, the exporter consists only of the smdexp.dle file. You do not need to export .vph files with this version of the exporter.

Index