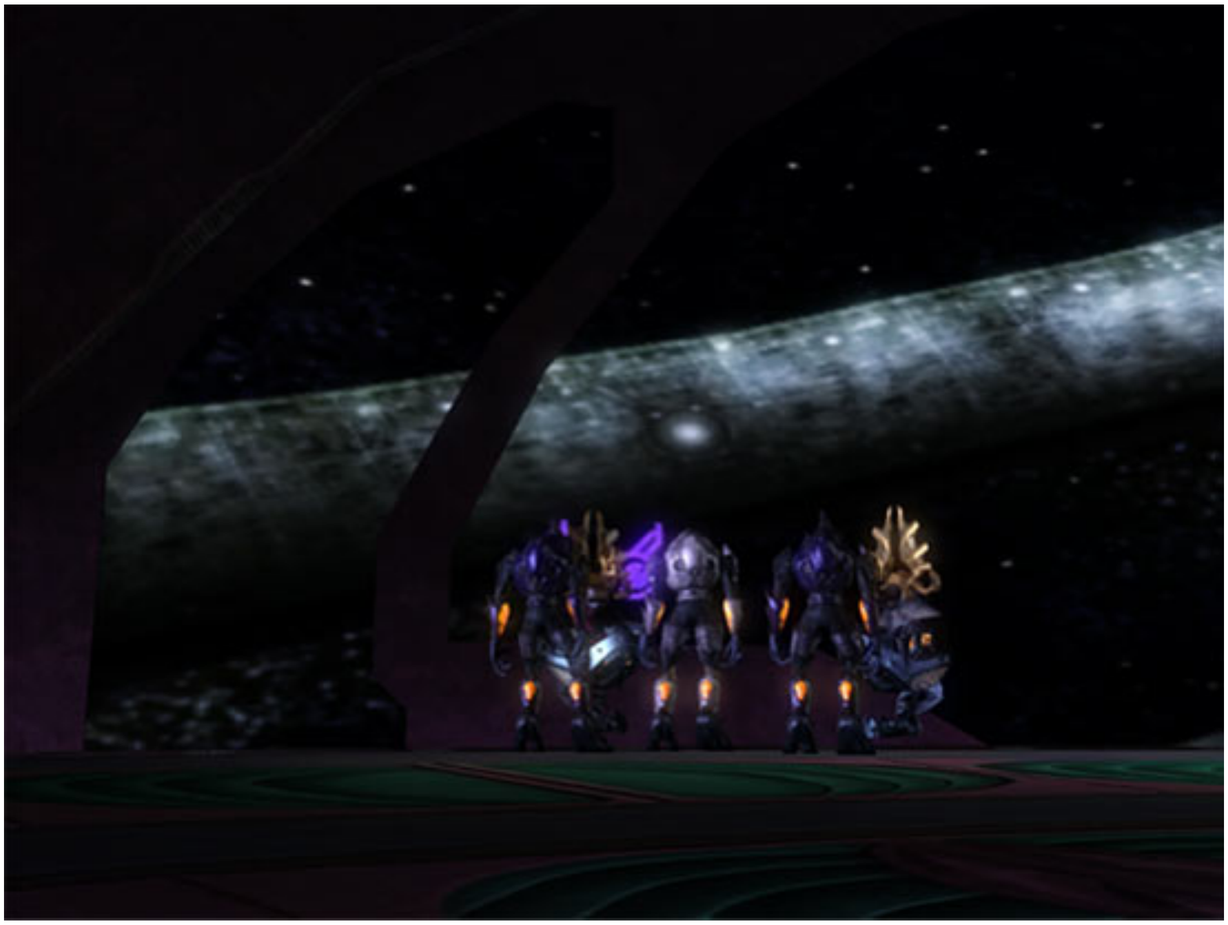
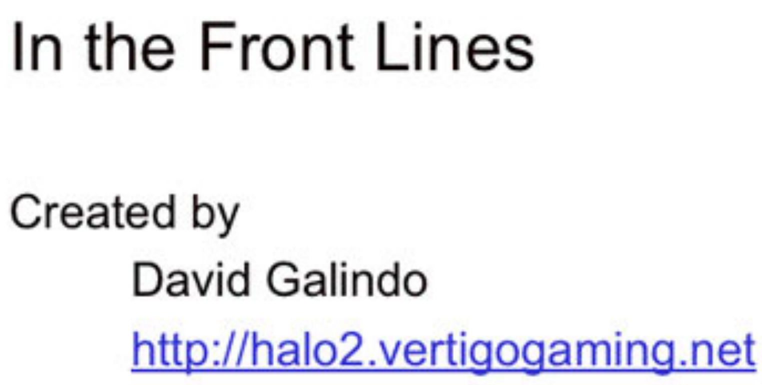
Containing the Flood

Content Management for Halo 2 and Beyond

by Mat Noguchi



What is Halo? At its core, Halo is a world simulation. Instead of trying to force a rigid structure onto players that defines the only way to play a game, we try to give the player a giant sandbox and let them play as they want.



In the Front Lines

Created by

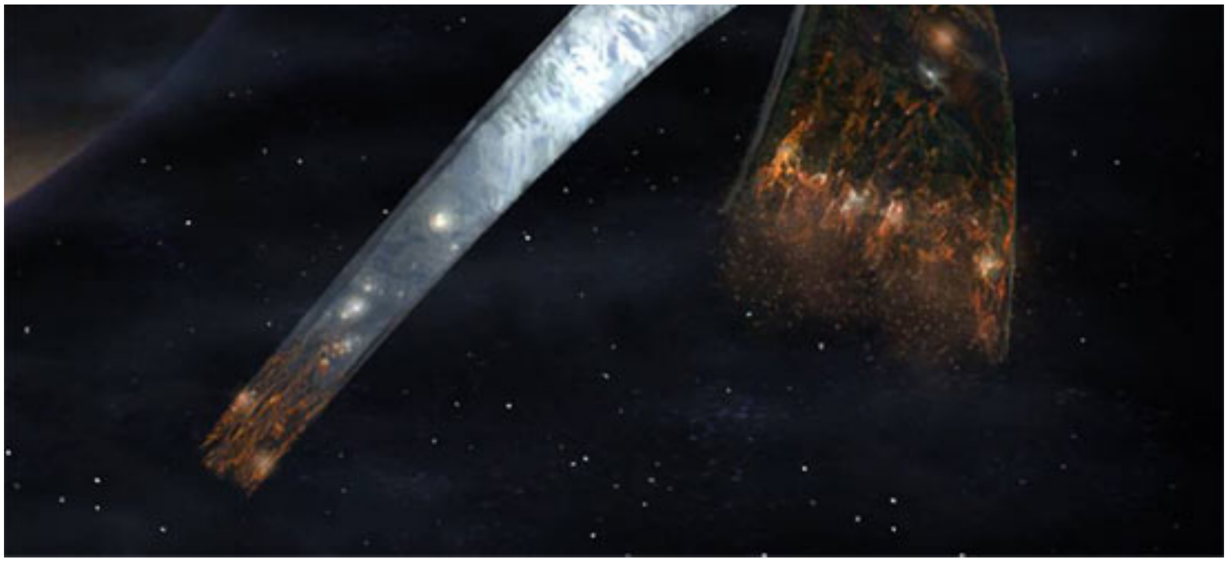
David Galindo

http://halo2.vertigogaming.net

The way we accomplish this is to treat the player as is simply another stimulus driving the overall state of the game. Ultimately, we don't really need the player to drive some of the more complex interactions in Halo. You could even remove the player from much of the game and still watch battles unfold between the Humans and Covenant. In fact, someone did just that.

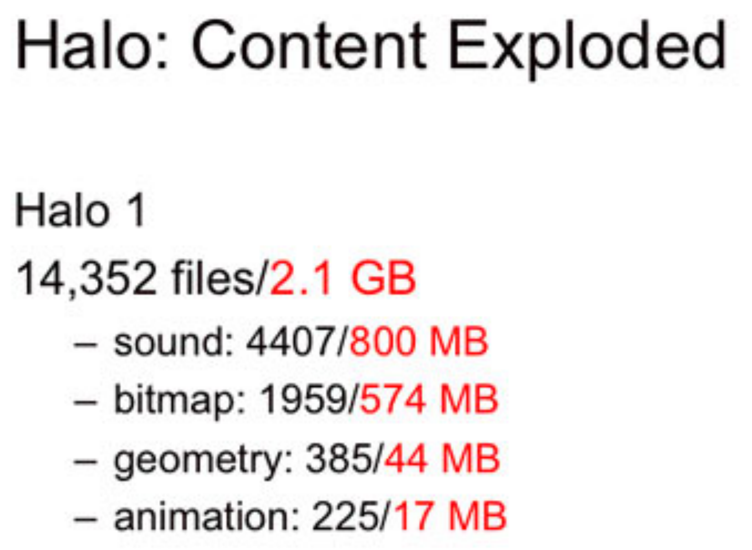
This is a fan made movie featuring the Marines. Credit goes to David Galindo for this beautiful expose of the Halo engine

Halo: Content Exploded



In order to even attempt to build this complete experience, we need to have not only a high level of detail, but a wide breadth of detail as well. It means that we need to make animations for every action a character can perform. It means we have to make effects for every material vs material interaction possible in the game, from plasma bolts hitting rusted metal or rocks hitting water to tires skidding on sand or purple metal grinding against wood. It means everything must make a sound, and every sound must have enough permutations to avoid repetition. It means that we need to have combat dialogue for every possible interaction between every speaking character in the game.

In other words, we need content.



Halo: Content Exploded

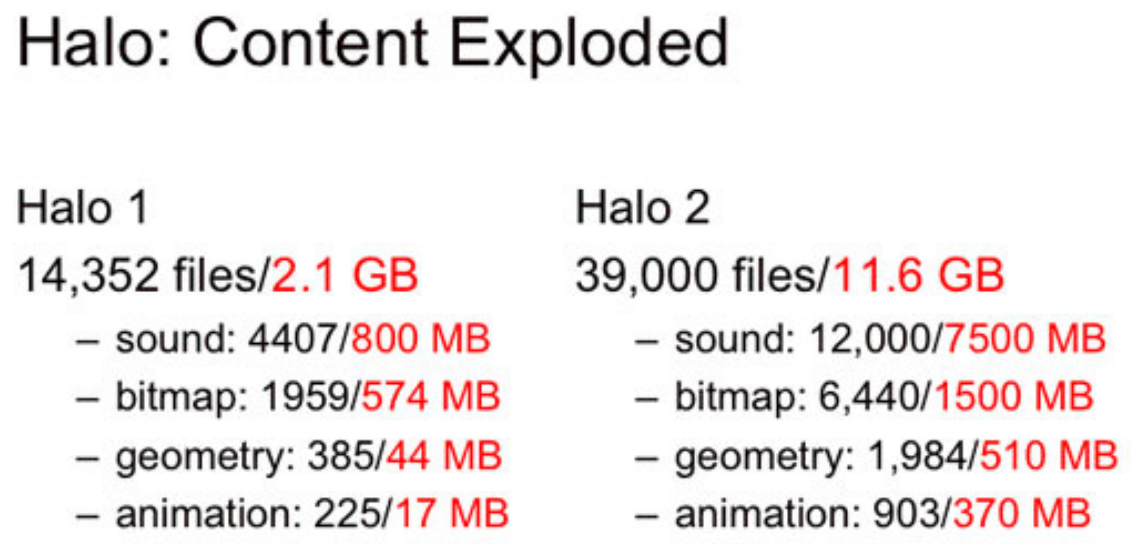
Halo 1

14,352 files/2.1 GB

- sound: 4407/800 MB
- bitmap: 1959/574 MB
- geometry: 385/44 MB
- animation: 225/17 MB

A lot of content.

These figures represent the total number of asset files needed to run the game excluding raw source data, like 3d Studio MAX files and photoshop textures. For Halo 1 we had a reasonable sized content tree. However, evolving our sandbox turned out to be a much bigger problem than we thought.



Halo: Content Exploded

Halo 1	Halo 2
14,352 files/2.1 GB	39,000 files/11.6 GB
– sound: 4407/800 MB	– sound: 12,000/7500 MB
– bitmap: 1959/574 MB	– bitmap: 6,440/1500 MB
– geometry: 385/44 MB	– geometry: 1,984/510 MB
– animation: 225/17 MB	– animation: 903/370 MB

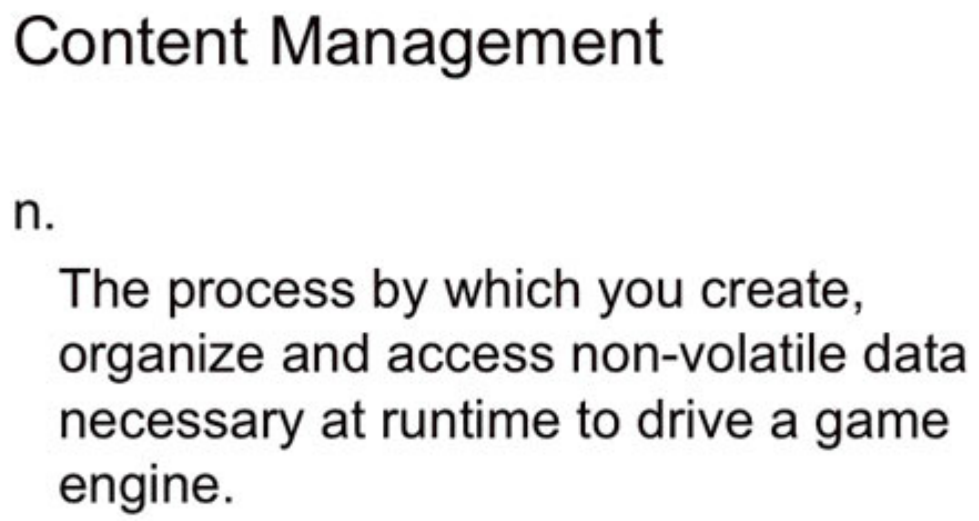
We saw an increase not only in the complexity of textures and geometry, which is to be expected with an upgraded graphics engine; we saw an even bigger explosion in the sheer amount of sound and animation data as well. In fact, when we integrated combat dialogue content into our game it added so much data that we had to rearchitect several layers of our resource system to handle the deluge of sound.

Increasing content demands can crush the game development process.

Clearly content will continue to be a major component of game development. It will only get worse as we increase the depth and breadth of the gameplay sandbox.

As the vanguard of game development, we programmers have the capability of meeting this challenge head on. We pretty much have to be, since we're the ones who take the content and use it to drive the game.

How do we do that? The answer is content managemen



Content Management

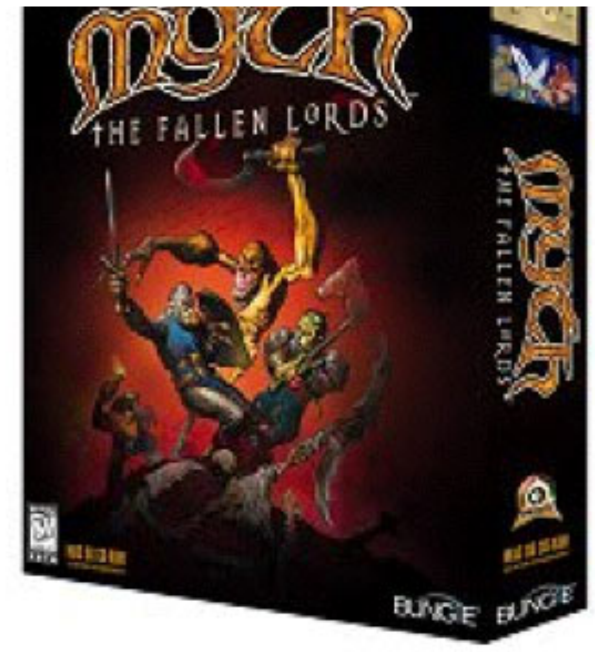
n.

The process by which you create, organize and access non-volatile data necessary at runtime to drive a game engine.

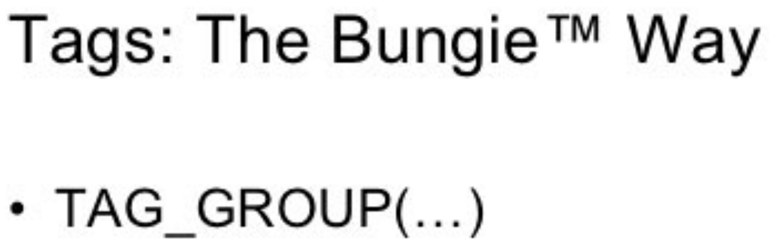
In simpler terms, programmatically managing game assets. In my professional opinion, content management was one of the most important aspects of Halo 2's development. That may be because that's what I get paid for, but that's besides the point.

For Halo 2, we were fortunate enough to have a system that handled all our resource management. We call it the tag system.

Tags: The Bungie™ Way



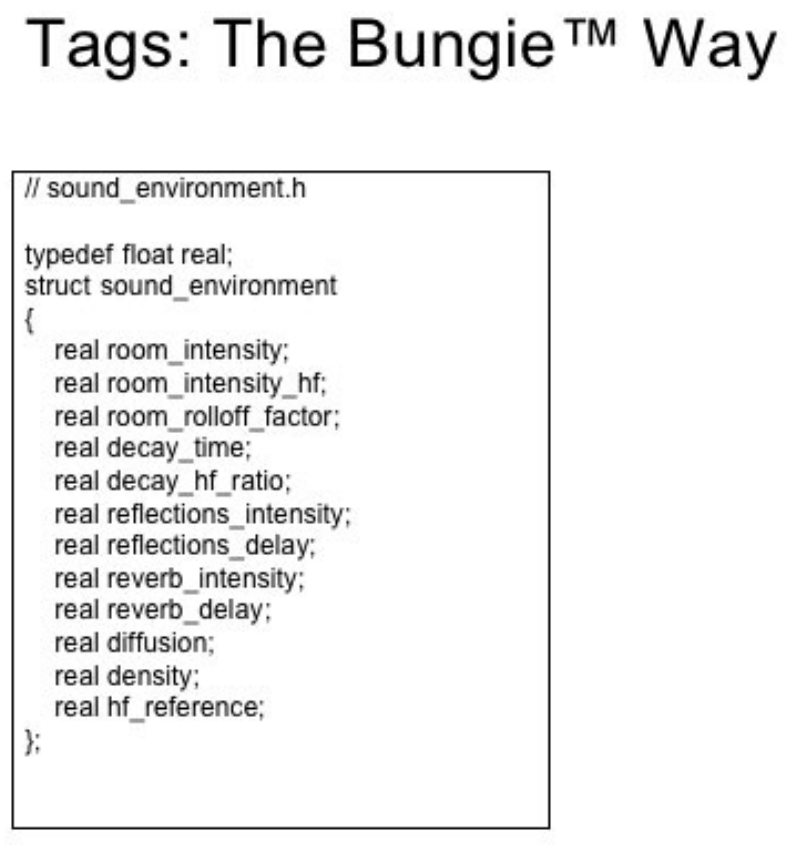
The tag system has its origins from Myth as a platform agnostic resource system, but it has evolved extensively since then.



Tags: The Bungie™ Way

- TAG_GROUP(…)

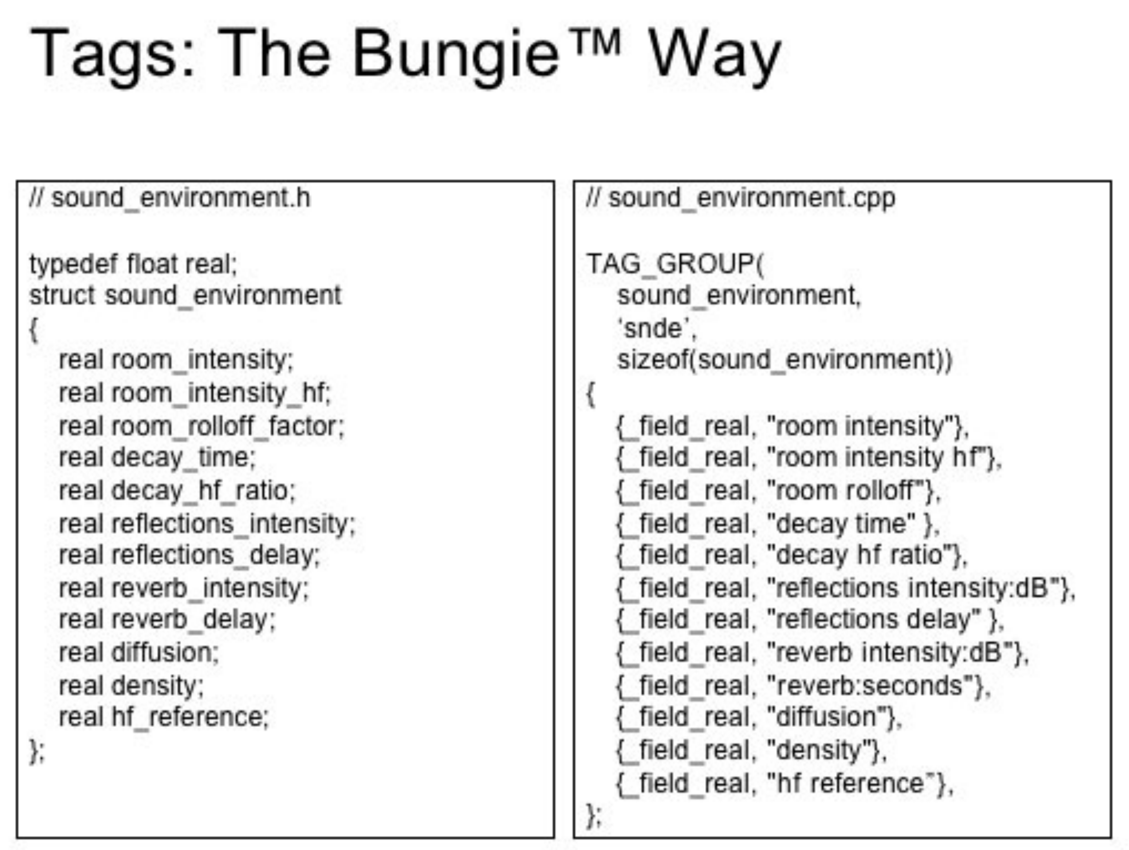
At its core, the tag system defines a set of interfaces and types that we use to manage C structs in a generic fashion. We do this by creating a schema in code for every C struct we wish to manage via the tag system; these schemas are called tag_groups.



Tags: The Bungie™ Way

```
// sound_environment.h

typedef float real;
struct sound_environment
{
   real room_intensity;
   real room_intensity_hf;
   real room_rolloff_factor;
   real decay_time;
   real decay_hf_ratio;
   real reflections_intensity;
   real reflections_delay;
   real reverb_intensity;
   real reverb_delay;
   real diffusion;
   real density;
   real hf_reference;
};
```

For a simple example, let's take a look at our sound_environment tag. In code, we have a sound_environment struct. To expose this struct to the tag system, we define a TAG_GROUP in C with a little bit of preprocessor magic like so. The tag_group definition provides the tag system with a straightforward mechanism for type reflection; we use reflection to perform universal tasks on tags, such as automatic serialization or building up editors.

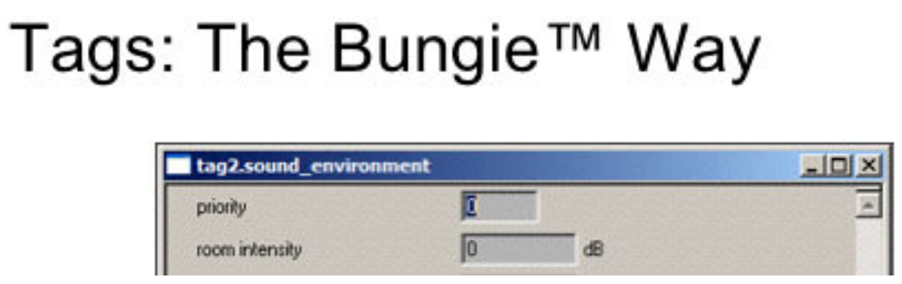


Tags: The Bungie™ Way

```
// sound_environment.h

typedef float real;
struct sound_environment
{
   real room_intensity;
   real room_intensity_hf;
   real room_rolloff_factor;
   real decay_time;
   real decay_hf_ratio;
   real reflections_intensity;
   real reflections_delay;
   real reverb_intensity;
   real reverb_delay;
   real diffusion;
   real density;
   real hf_reference;
};
```

```
// sound_environment.cpp

TAG_GROUP(
   sound_environment,
   'snde',
   sizeof(sound_environment))
{
   {_field_real, "room intensity"},
   {_field_real, "room intensity hf"},
   {_field_real, "room rolloff"},
   {_field_real, "decay time" },
   {_field_real, "decay hf ratio"},
   {_field_real, "reflections intensity:dB"},
   {_field_real, "reflections delay" },
   {_field_real, "reverb intensity:dB"},
   {_field_real, "reverb:seconds"},
   {_field_real, "diffusion"},
   {_field_real, "density"},
   {_field_real, "hf reference"},
};
```

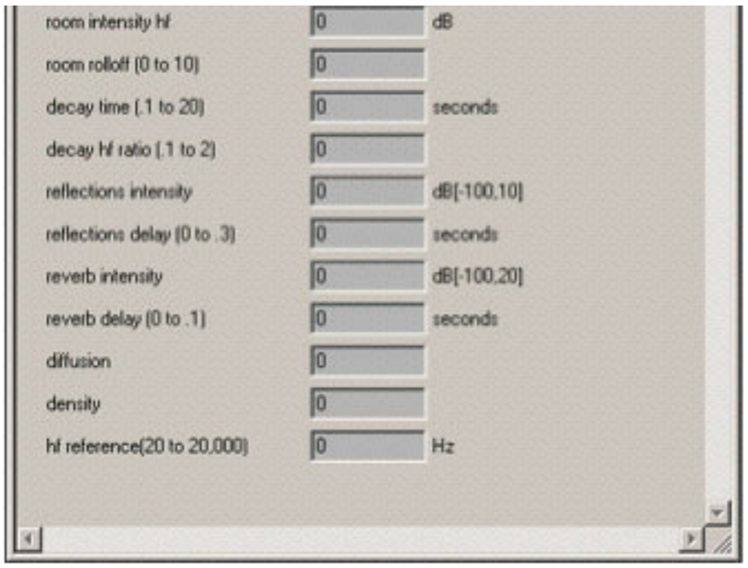
For a simple example, let's take a look at our sound_environment tag. In code, we have a sound_environment struct. To expose this struct to the tag system, we define a TAG_GROUP in C with a little bit of preprocessor magic like so. The tag_group definition provides the tag system with a straightforward mechanism for type reflection; we use reflection to perform universal tasks on tags, such as automatic serialization or building up editors.



Tags: The Bungie™ Way

tag2.sound_environment

priority	0	
room intensity	0	dB



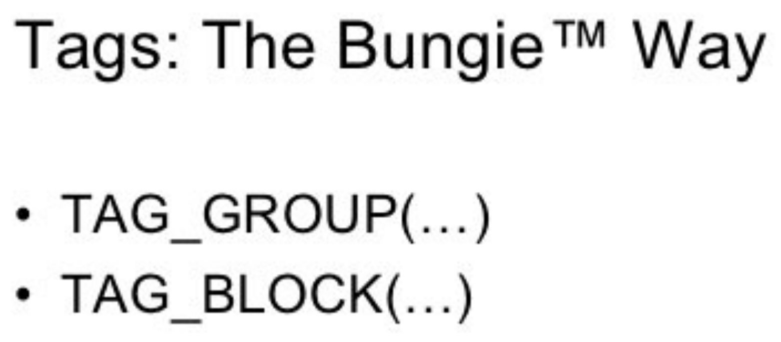
room intensity hf	0	dB
room rolloff (0 to 10)	0	
decay time (.1 to 20)	0	seconds
decay hf ratio (.1 to 2)	0	
reflections intensity	0	dB[-100,10]
reflections delay (0 to .3)	0	seconds
reverb intensity	0	dB[-100,20]
reverb delay (0 to .1)	0	seconds
diffusion	0	
density	0	
hf reference(20 to 20,000)	0	Hz

This is how the sound_environment tag would be edited in Guerilla, our tag editor.

Tags: The Bungie™ Way

- TAG_GROUP(…)
  - tag_group *tag_group_get(
    tag group_tag)
  - long tag_load(
    const char *name,
    tag group_tag)
  - void *tag_get(
    long tag_index,
    tag group_tag)

These are a few of the functions that we use to access and manipulate tags. The most important of these is tag_get(...). It allows us to represent resources universally as handles, as well as providing a standard interface for accessing tags. I'll revisit the importance of this later on.

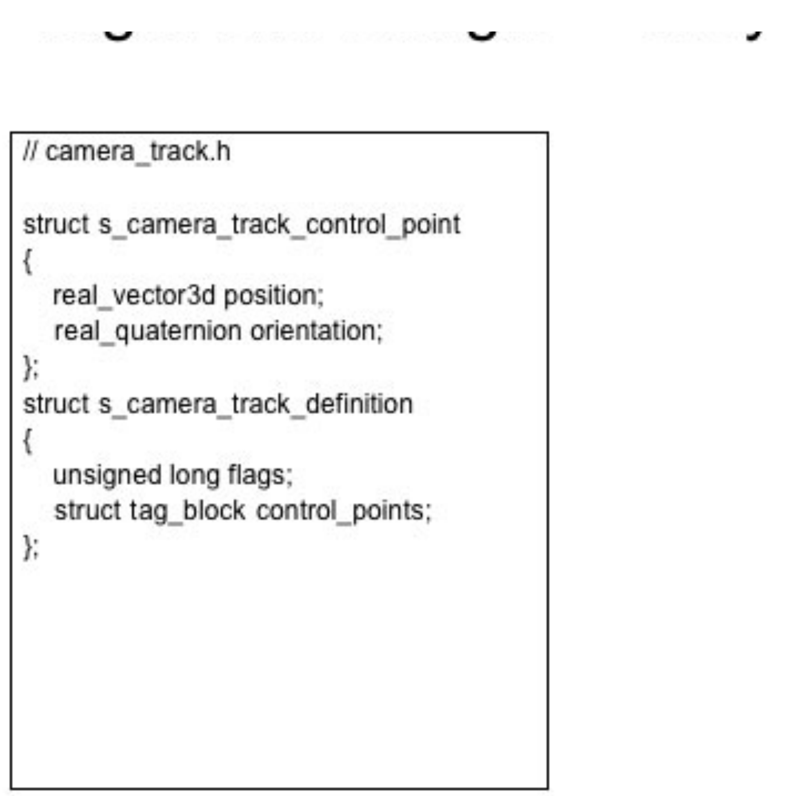


Tags: The Bungie™ Way

- TAG_GROUP(…)
- TAG_BLOCK(…)

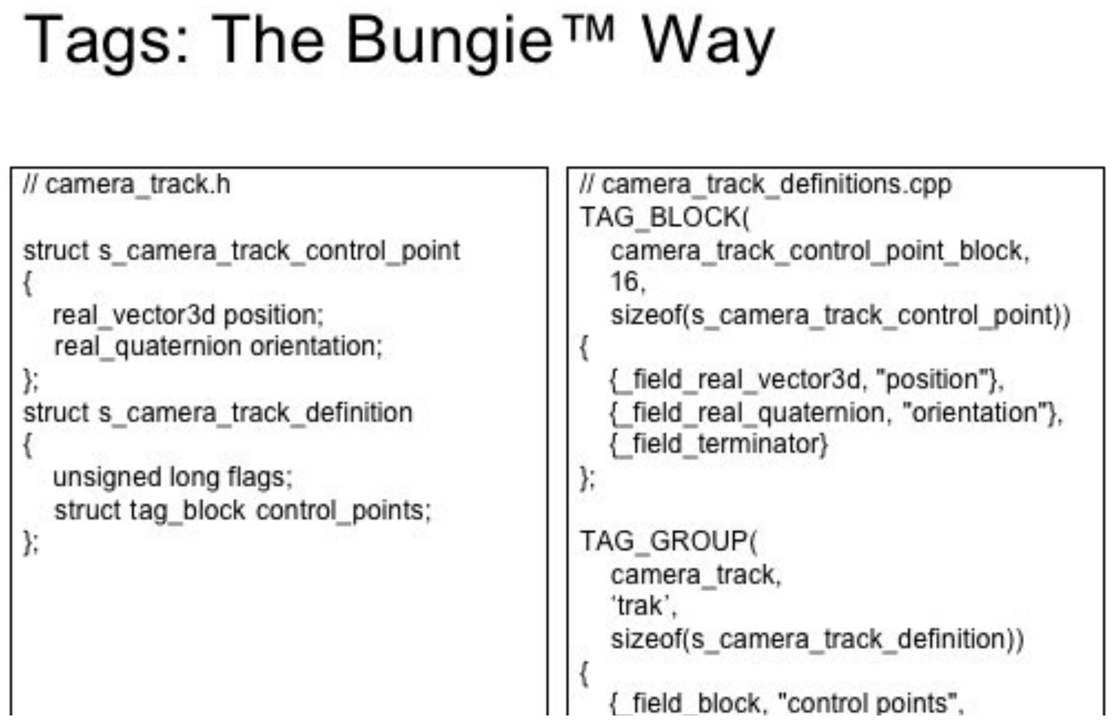
To allow for more complex data structures than a simple C struct, we also have a type called a **tag_block**. A tag_block is similar to a tag_group; it defines a resizeable array whose elements correspond to a C struct. Since both a tag_group and tag_block can contain any set of fields, including tag_blocks, we can describe almost any type of structured data hierarchy. By providing a generic container type, we also push the responsibility of resource memory management exclusively into the realm of the tag system.

Tags: The Bungie™ Way



```
// camera_track.h

struct s_camera_track_control_point
{
   real_vector3d position;
   real_quaternion orientation;
};
struct s_camera_track_definition
{
   unsigned long flags;
   struct tag_block control_points;
};
```

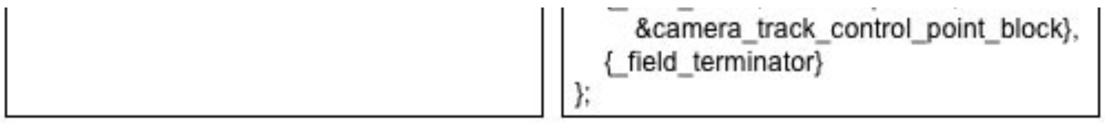
As an example, this is how we define our camera track tag. Here's the C struct definition. Followed by the tag_group definition.



Tags: The Bungie™ Way

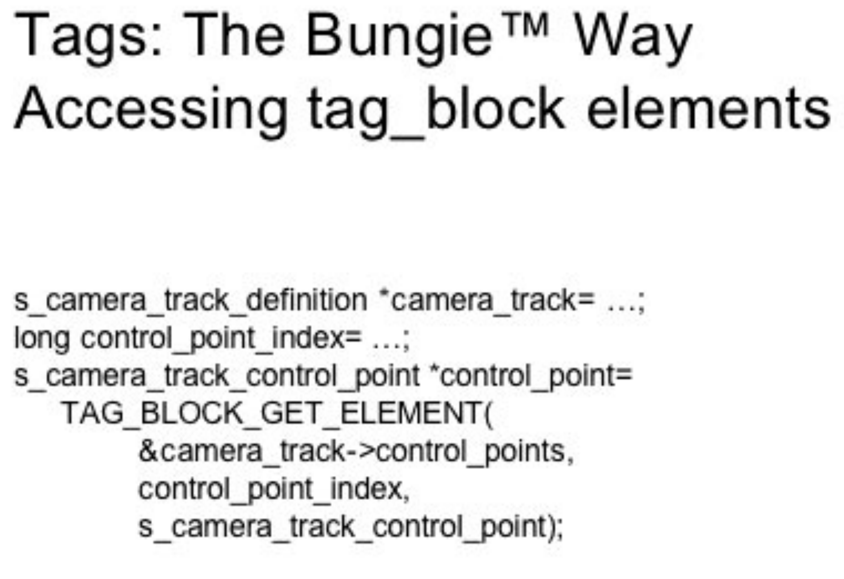
```
// camera_track.h

struct s_camera_track_control_point
{
   real_vector3d position;
   real_quaternion orientation;
};
struct s_camera_track_definition
{
   unsigned long flags;
   struct tag_block control_points;
};
```

```
// camera_track_definitions.cpp
TAG_BLOCK(
   camera_track_control_point_block,
   16,
   sizeof(s_camera_track_control_point))
{
   {_field_real_vector3d, "position"},
   {_field_real_quaternion, "orientation"},
   {_field_terminator}
};

TAG_GROUP(
   camera_track,
   'trak',
   sizeof(s_camera_track_definition))
{
   {_field_block, "control points",
```



```
        &camera_track_control_point_block},
    {_field_terminator}
};
```

As an example, this is how we define our camera track tag. Here's the C struct definition. Followed by the tag_group definition.

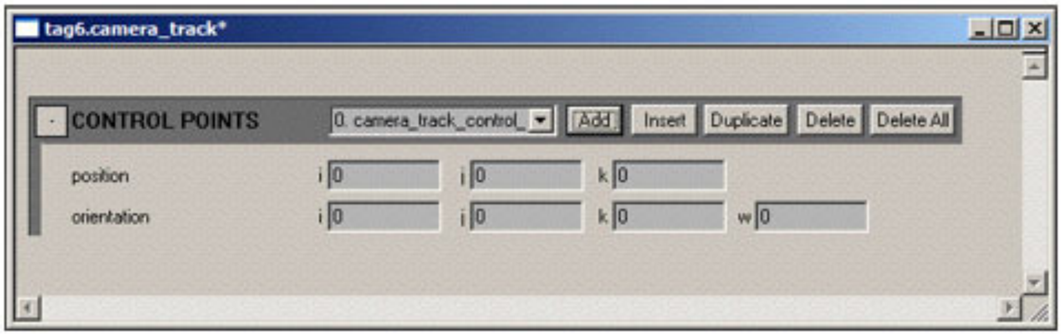


Tags: The Bungie™ Way Accessing tag_block elements

```
s_camera_track_definition *camera_track= …;
long control_point_index= …;
s_camera_track_control_point *control_point=
    TAG_BLOCK_GET_ELEMENT(
            &camera_track->control_points,
            control_point_index,
            s_camera_track_control_point);
```

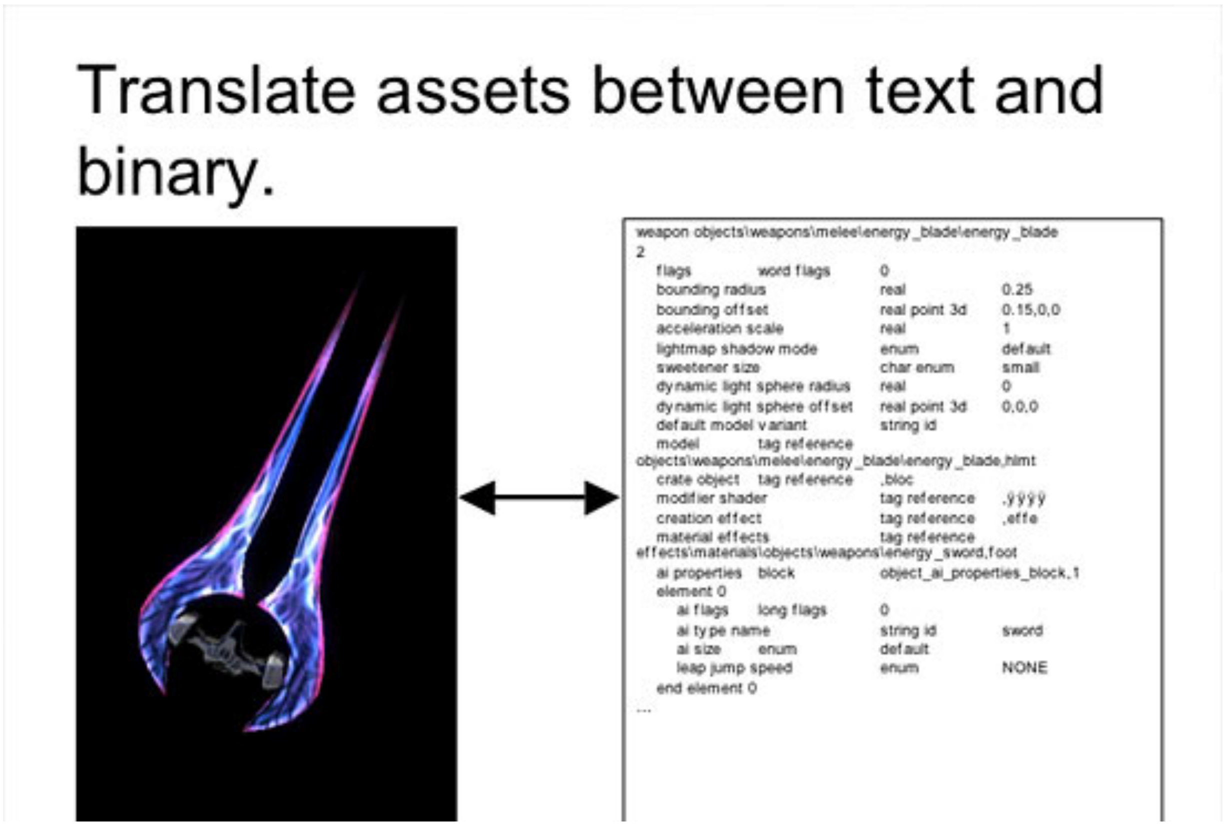
To access elements of a tag_block, you have to use a macro. However, this is an implementation detail that can easily be abstracted out via more preprocessor macros or through templated wrappers on top of the tag_block structure.

Tags: The Bungie™ Way



```
position       i 0    j 0    k 0
orientation    i 0    j 0    k 0    w 0
```

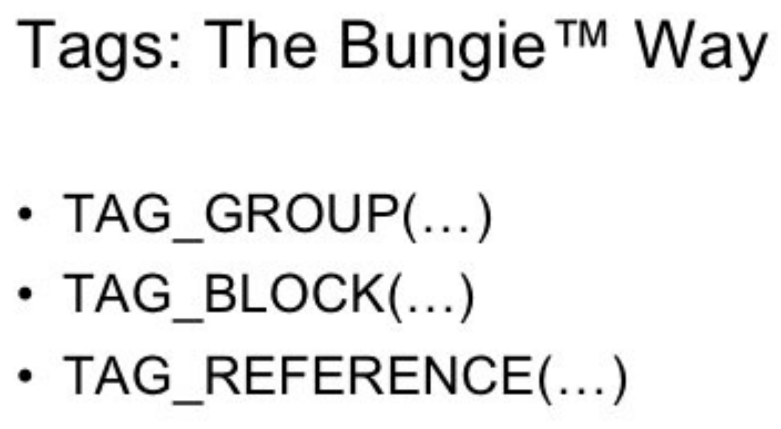
And here's how it would be presented in Guerilla. As you can see, we have a simple interface for manipulating the tag_block as an array of elements.



Translate assets between text and binary.

```
weapon objects\weapons\melee\energy_blade\energy_blade
2
   flags          word flags          0
   bounding radius                    real              0.25
   bounding offset                    real point 3d     0.15,0,0
   acceleration scale                 real              1
   lightmap shadow mode               enum              default
   sweetener size                     char enum         small
   dynamic light sphere radius        real              0
   dynamic light sphere offset        real point 3d     0,0,0
   default model variant              string id
   model          tag reference
objects\weapons\melee\energy_blade\energy_blade,hlmt
   crate object   tag reference       ,bloc
   modifier shader                    tag reference     ,ÿÿÿÿ
   creation effect                    tag reference     ,effe
   material effects                   tag reference
effects\materials\objects\weapons\energy_sword,foot
   ai properties  block               object_ai_properties_block,1
   element 0
      ai flags    long flags          0
      ai type name                    string id         sword
      ai size     enum                default
      leap jump speed                 enum              NONE
   end element 0
...
```

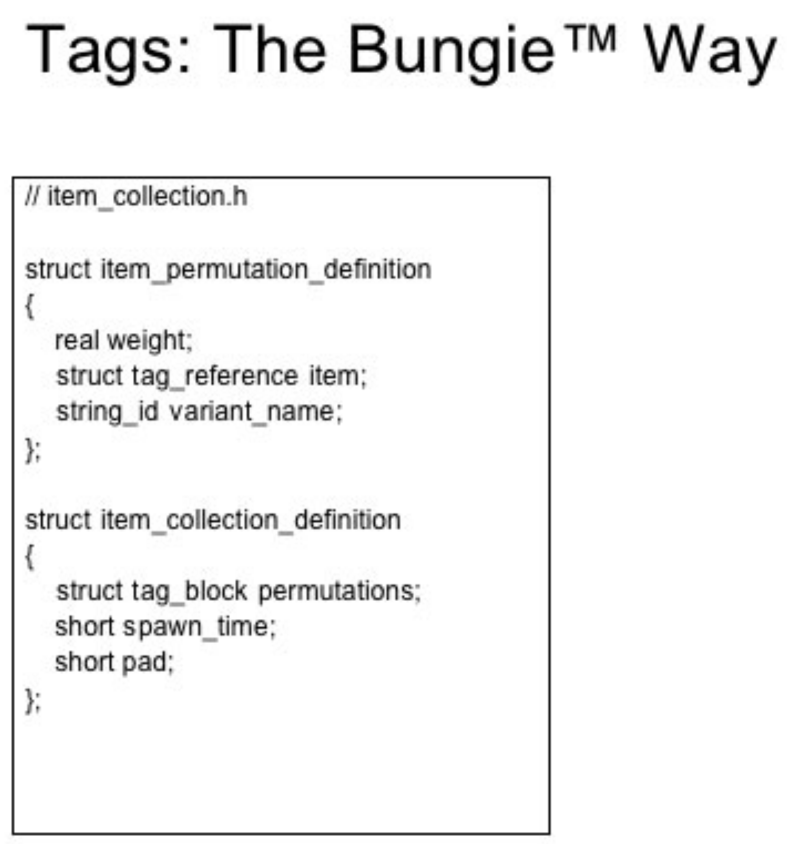
We can also walk the tag definition to translate between various tag representations; in this case, we can take the raw binary data for the energy sword tag and translate it to and from text with ease. We don't do this operation often, but it's useful to analyze how content changed in source control. We had an intern go in and add this capability to our source control process late in the project; it helped us immensely in tracking changes, especially as we started locking down the content and instituted a peer-review checkin policy.



Tags: The Bungie™ Way

- TAG_GROUP(…)
- TAG_BLOCK(…)
- TAG_REFERENCE(…)

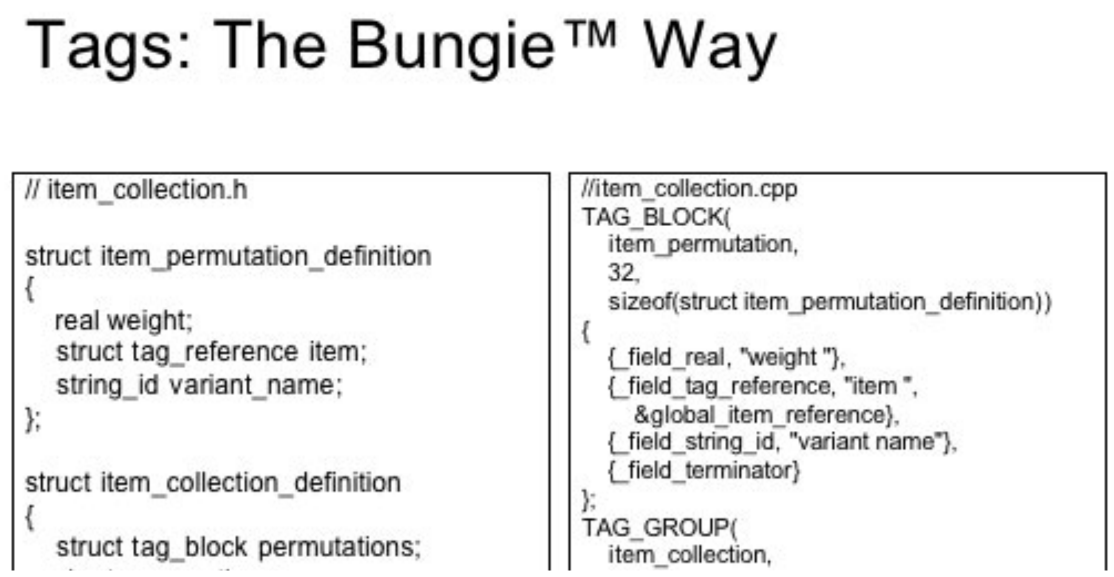
We also have a type to allow tags to explicitly define external dependencies, called a **tag_reference**. A tag_reference is essentially a typed persistent pointer that is maintained by the tag system when a tag is loaded. A tag_reference can be further constrained to point to tags of a particular tag_group. This simple abstraction is what powers much of our game asset management.



Tags: The Bungie™ Way

```
// item_collection.h

struct item_permutation_definition
{
   real weight;
   struct tag_reference item;
   string_id variant_name;
};

struct item_collection_definition
{
   struct tag_block permutations;
   short spawn_time;
   short pad;
};
```

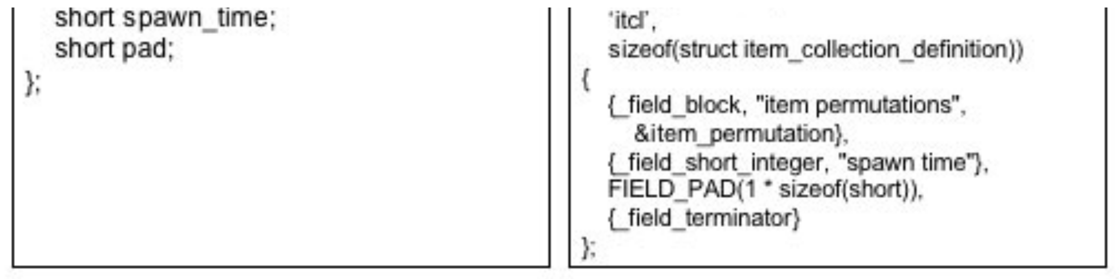
Here's a tag that uses tag_references, the item_collection_definition struct. The multiplayer engine uses this tag to determine what kinds of items or weapons can spawn at a particular location.



Tags: The Bungie™ Way

```
// item_collection.h

struct item_permutation_definition
{
   real weight;
   struct tag_reference item;
   string_id variant_name;
};

struct item_collection_definition
{
   struct tag_block permutations;
```

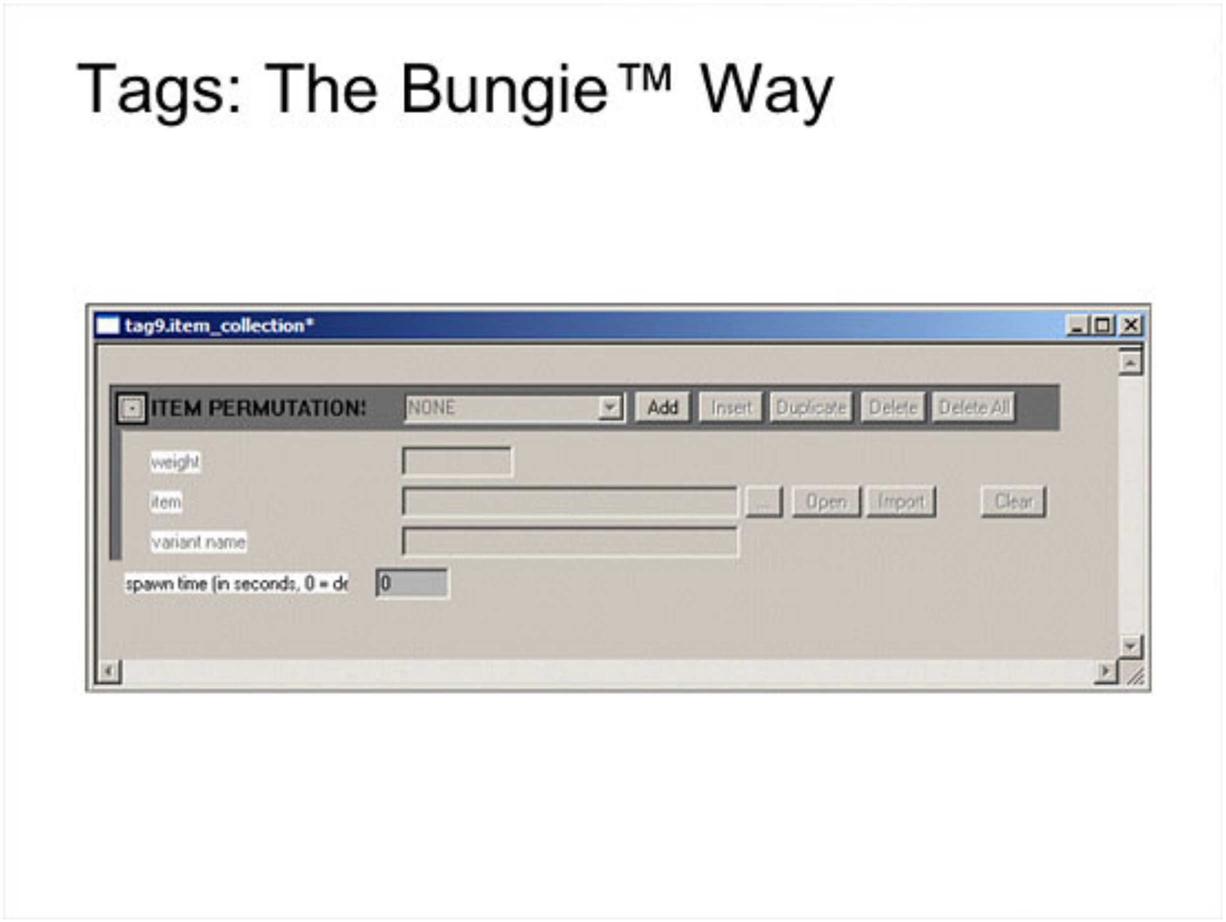
```
//item_collection.cpp
TAG_BLOCK(
   item_permutation,
   32,
   sizeof(struct item_permutation_definition))
{
   {_field_real, "weight "},
   {_field_tag_reference, "item ",
      &global_item_reference},
   {_field_string_id, "variant name"},
   {_field_terminator}
};
TAG_GROUP(
   item_collection,
```



```
    short spawn_time;
    short pad;
};
```

```
    'itcl',
    sizeof(struct item_collection_definition))
{
    {_field_block, "item permutations",
        &item_permutation},
    {_field_short_integer, "spawn time"},
    FIELD_PAD(1 * sizeof(short)),
    {_field_terminator}
};
```

Here's a tag that uses tag_references, the item_collection_definition struct. The multiplayer engine uses this tag to determine what kinds of items or weapons can spawn at a particular location.



Tags: The Bungie™ Way

tag9.item_collection*

ITEM PERMUTATION: NONE

weight

item

variant name

spawn time (in seconds, 0 = dε 0

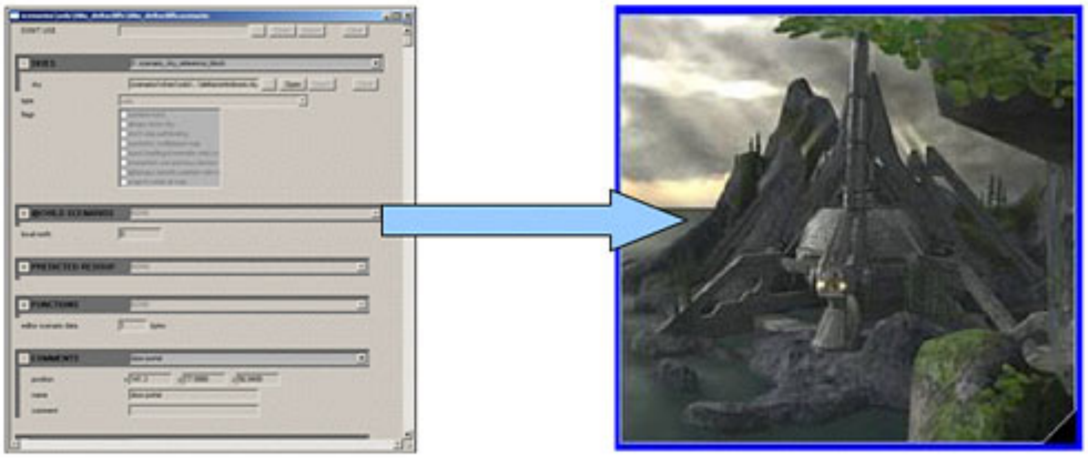
From this definition we generate the following interface in Guerilla. To edit a tag_reference, we simply invoke the common file dialog from Windows.

With explicitly defined external references, we can analyze and operate on complex resource hierarchies as a directed graph. We use this graph representation when loading tags for gameplay; to load a tag, we walk its dependency graph using a depth first search and load each tag as we go. The tag_reference encourages design through composition; it is very easy to connect game assets together by editing tag_references, and it allows for quick prototyping from existing content.

For example, this is how we created the warthog variants; the chaingun and gauss turrets were created as separate objects and attached to the warthog as opposed to being modeled as part of the warthog.

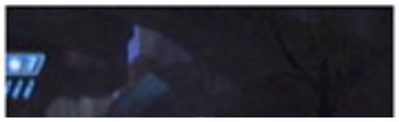
This is also the typical method used to hack Halo on the Xbox; by directly manipulating tag references.

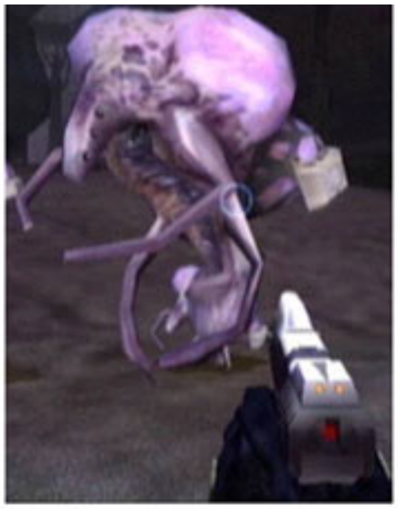
Load levels from a single resource.

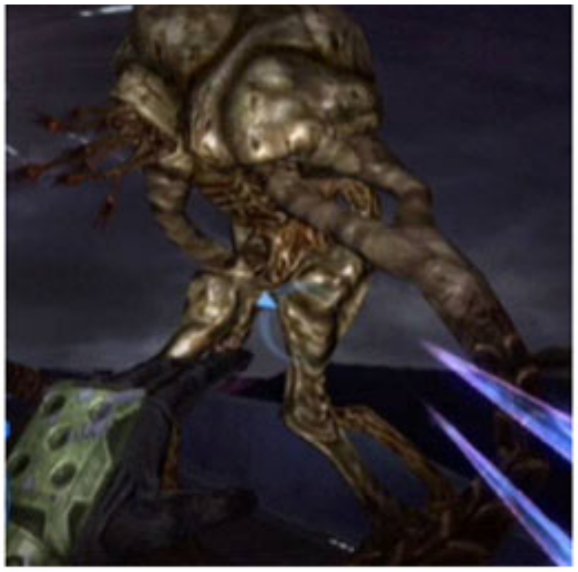


With most of our game assets defined as tags, none of our leaf game systems (e.g., physics, sound, rendering, or AI) manage their respective assets; all resources are handled by the tag system; tags must be accessed through handles, and all tag memory is managed by the tag system. We also enforce a read-only view of tag data for most game systems; the main exception being physics due to how we integrated Havok into our engine. To facilitate this separation of control between the game and the tag system, we only explicitly load two tags: the **globals** tag and the **scenario** tag. The scenario tag is the level representation. The globals and scenario tags are available for any game system to access, so for a game system to have access to a particular tag, it must be accessible through the globals tag or the scenario tag.

OOPS!

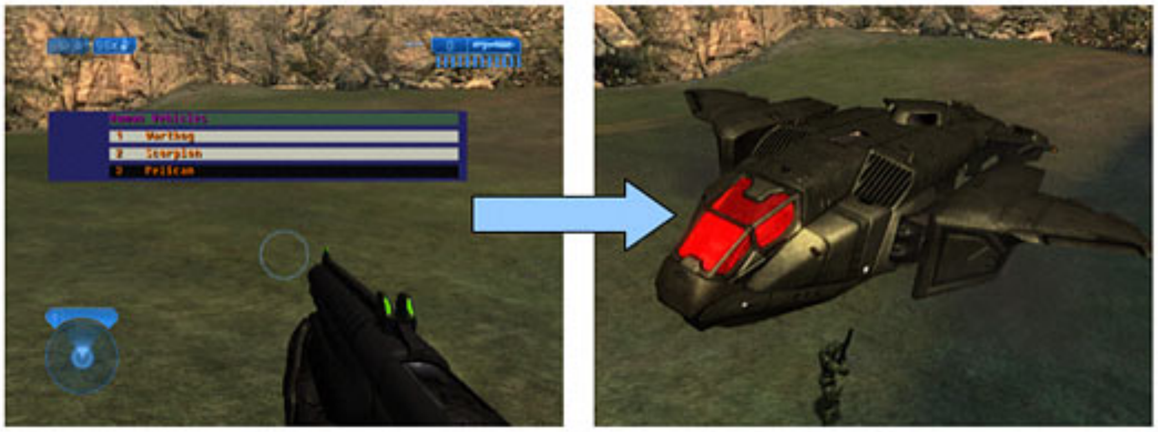






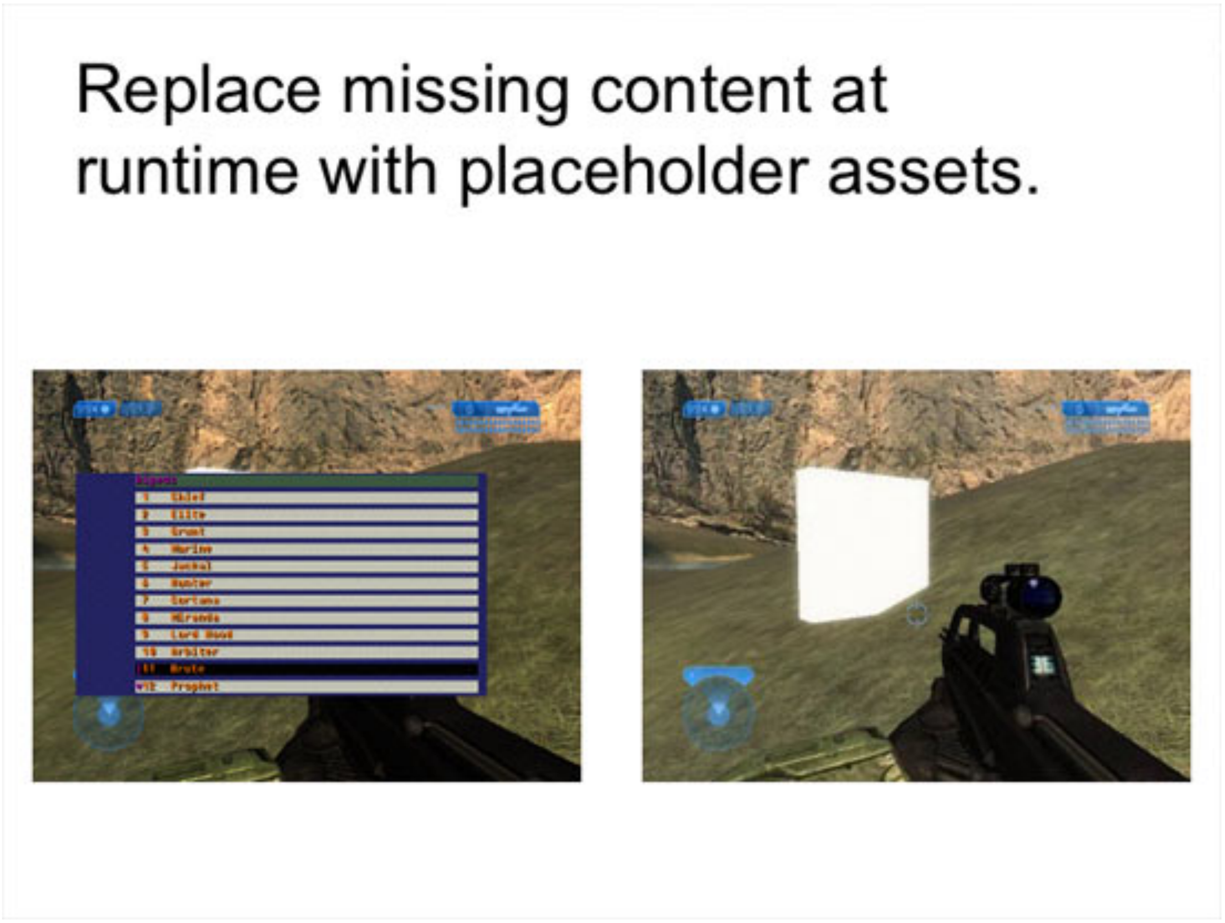
However, this means that occasionally tags get referenced into a particular level and are then forgotten. This is how we managed to ship Halo 1 with the Engineer, and Halo 2 with the Flood Juggernaut, two characters that were cut before we shipped.

Using scripts to work around singular loading process.



There is another unfortunate consequence of this singular level loading behavior. Because the only supported way to run the game requires a level to load, it has been very difficult for us to develop interactive tools for viewing and editing data that doesn't need a level loaded, e.g., interactively editing particle effects or viewing animations. What usually happened is that we added scripting commands in the game engine to manually load a given tag and use the automatic reloads to update it on the fly.

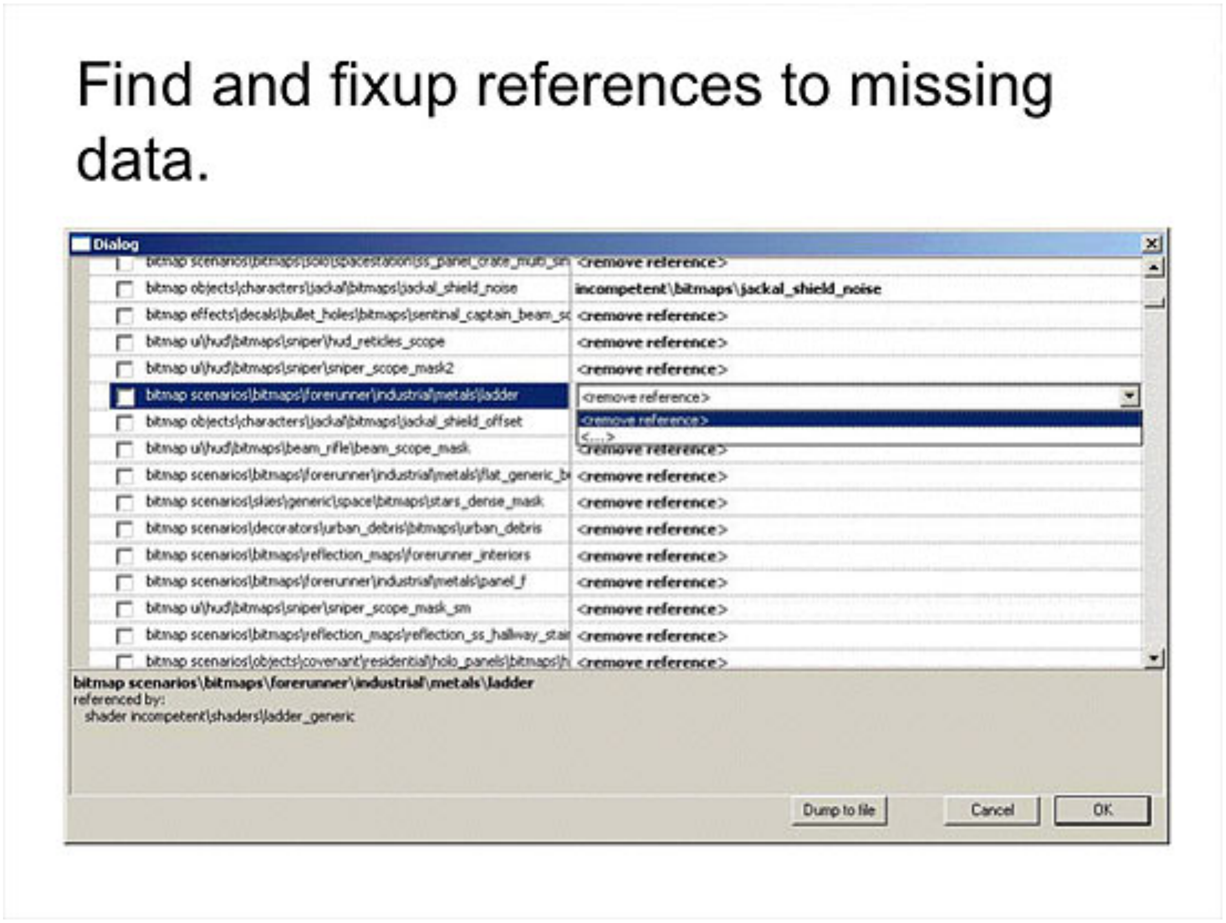
<Demo of drop object.>



Replace missing content at runtime with placeholder assets.

We can also take advantage of a standard recursive load process by customizing load behavior depending on various runtime conditions. One helpful thing to do is to substitute missing tags at runtime. If we cannot find or load a specific tag, we can instead load an appropriate placeholder tag; this helps in many ways, the most obvious is that it is easy to identify missing content.

We made this a default property of the tag loading process. For example, if I try to drop an object tag that does not exist, the game will drop this placeholder object instead.



Find and fixup references to missing data.

Dialog

bitmap scenarios\bitmaps\solo\spacestation\ss_panel_crate_multi_sm	<remove reference>
bitmap objects\characters\jackal\bitmaps\jackal_shield_noise	**incompetent\bitmaps\jackal_shield_noise**
bitmap effects\decals\bullet_holes\bitmaps\sentinal_captain_beam_sc	<remove reference>
bitmap ui\hud\bitmaps\sniper\hud_reticles_scope	<remove reference>
bitmap ui\hud\bitmaps\sniper\sniper_scope_mask2	<remove reference>
bitmap scenarios\bitmaps\forerunner\industrial\metals\ladder	<remove reference>
bitmap objects\characters\jackal\bitmaps\jackal_shield_offset	<remove reference>
bitmap ui\hud\bitmaps\beam_rifle\beam_scope_mask	<remove reference>
bitmap scenarios\bitmaps\forerunner\industrial\metals\flat_generic_b	<remove reference>
bitmap scenarios\skies\generic\space\bitmaps\stars_dense_mask	<remove reference>
bitmap scenarios\decorators\urban_debris\bitmaps\urban_debris	<remove reference>
bitmap scenarios\bitmaps\reflection_maps\forerunner_interiors	<remove reference>
bitmap scenarios\bitmaps\forerunner\industrial\metals\panel_f	<remove reference>
bitmap ui\hud\bitmaps\sniper\sniper_scope_mask_sm	<remove reference>
bitmap scenarios\bitmaps\reflection_maps\reflection_ss_hallway_stair	<remove reference>
bitmap scenarios\objects\covenant\residential\holo_panels\bitmaps\h	<remove reference>

<remove reference>
<...>

bitmap scenarios\bitmaps\forerunner\industrial\metals\ladder
referenced by:
shader incompetent\shaders\ladder_generic

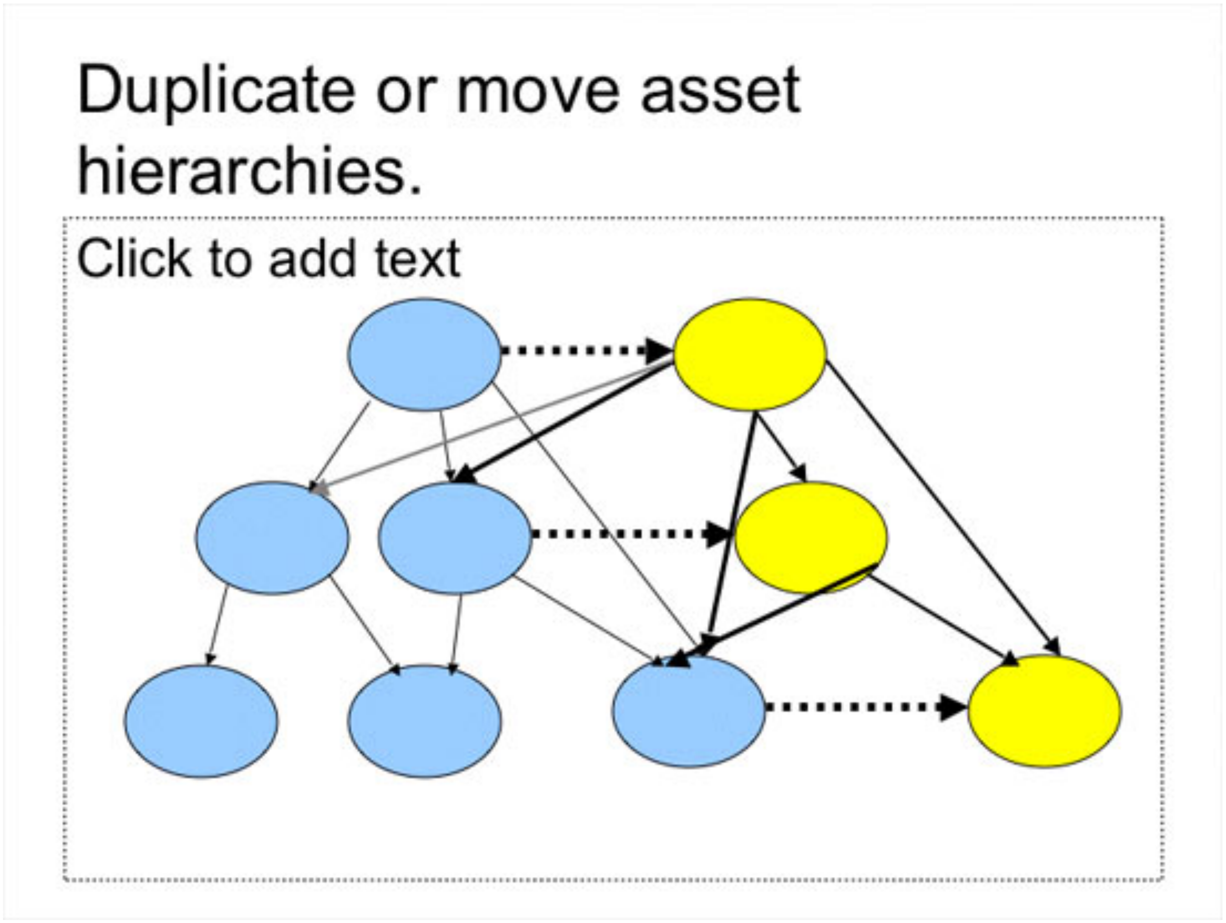
Replacing missing data at runtime is great, but let's say that I'm an art lead and I want to know just how much missing data there is in the content tree. If we load all the tags in the content tree and treat the tag_references as directed links, we can generate a graph and inspect that directly to find references to missing data.

<Demo of fix dependencies>

As you can see, the content tree I have here has a wide range of dangling references. If I wanted to, I could use this tool to redirect dangling references to existing content or clear them all-together. We can use the same information to determine simple data relationships such as, "How many tags refer to this blood splatter effect?"

<Demo of find referrent tags>

These are a couple of simple uses of graph analysis of content.



Duplicate or move asset hierarchies.

Click to add text

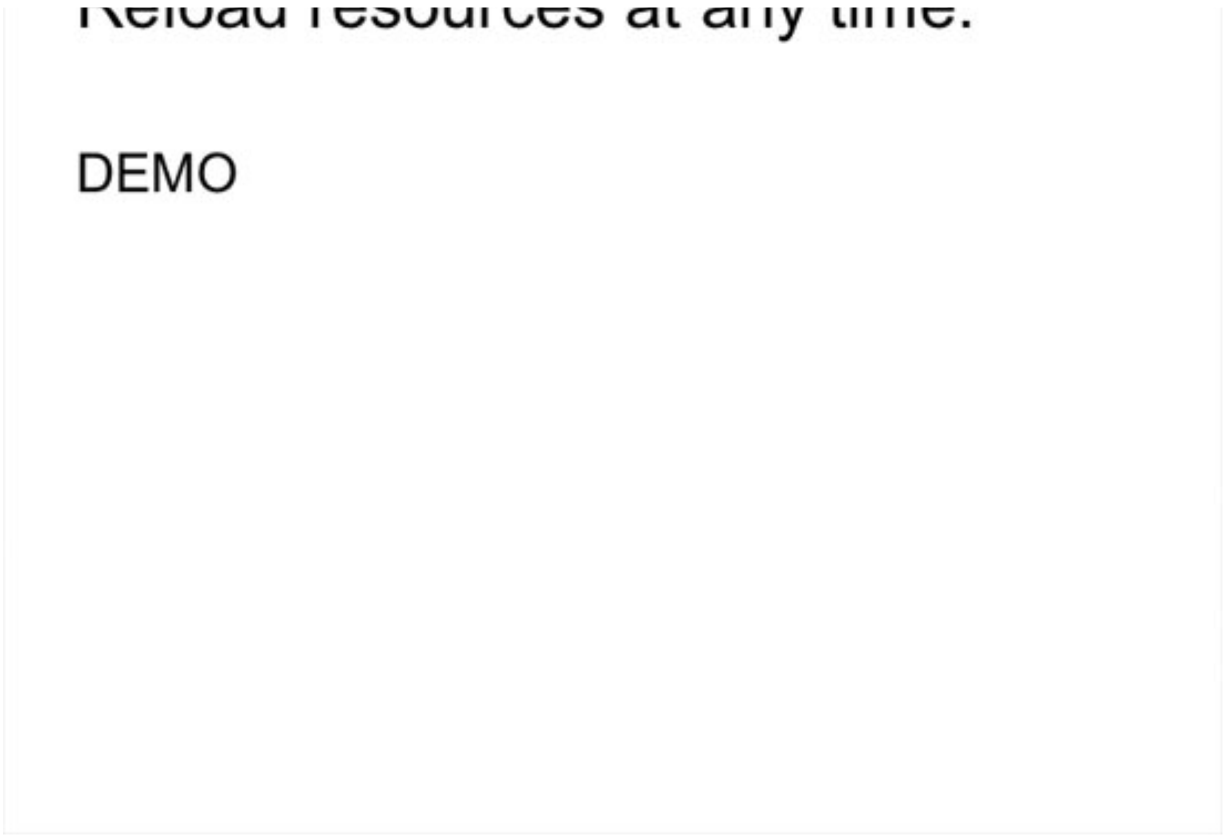
We can also use the dependency graph to perform more complex content operations. A quick way to prototype new gameplay ideas is to create a copy of an existing tag and tweak various values. I can accomplish this easily by just copying the file through Explorer, but what if I want to modify a whole slew of values across the tag hierarchy? What if I want to create a new variation of the warthog?

If I generate a dependency graph between a selection of tags, I can copy the tags to a new location and fixup all the references between the selected tags.

<Demo of copy tags.>

On a more global scale, we can use the dependency graph to move tags around the file system without breaking existing dependencies. We have done this on more than one occasion to migrate to new content directory schemes.

Reload resources at any time

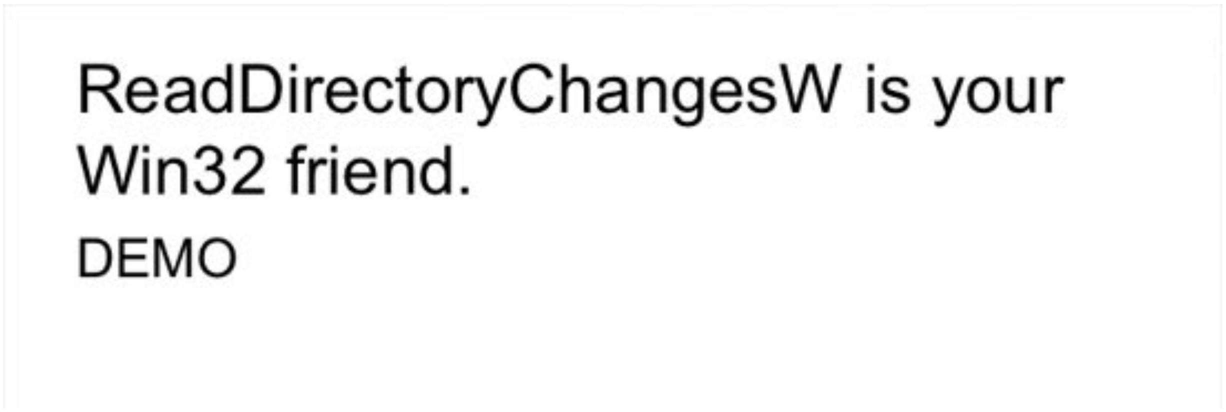

Reload resources at any time.

DEMO

Since we only allow access to tag data through handles provided by the tag system, any tag can be reloaded or modified at any time during the lifetime of a game quickly and easily, without adversely affecting other game systems. To make this system more robust, we have a single entry point for reloading tags in case we have a need to have custom cleanup or initialization code when a tag reloads.

This behavior is critical for artist iteration. To demonstrate the power of this system, we're going to go upgrade a few weapons.

<Weapon demo>


ReadDirectoryChangesW is your Win32 friend.

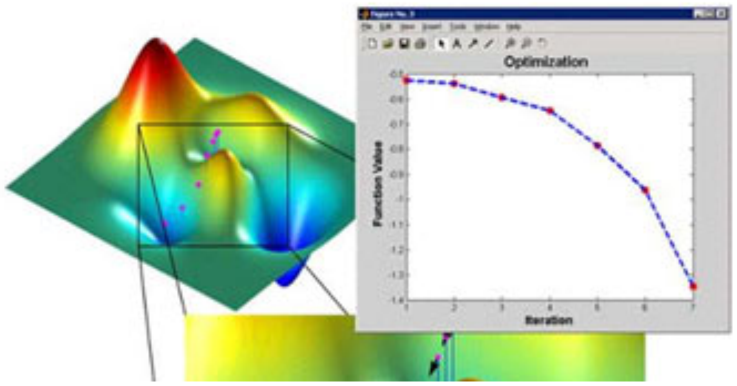
DEMO

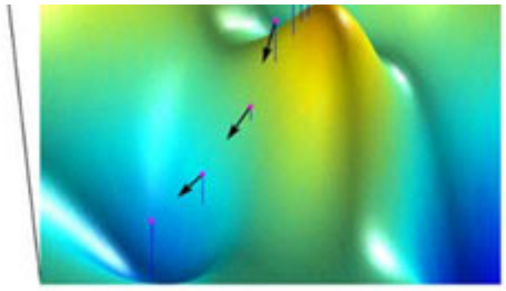
I'm sure readers of Game Developer have heard about our super magical monitoring tools. For those who haven't, we have tools that monitor the file system and reimport and synchronize content on the fly. We accomplish this little piece of magic using ReadDirectoryChangesW, a function that lets you monitor changes on a directory or volume. This is a very brute force way to propagate content changes, but it allows us to change what programs we create content in without having to rewrite export plugins.

To demonstrate this powerful tool, I'm going to show you how we would go about creating a new vehicle from an existing one.

<Demo of yellow warthog>

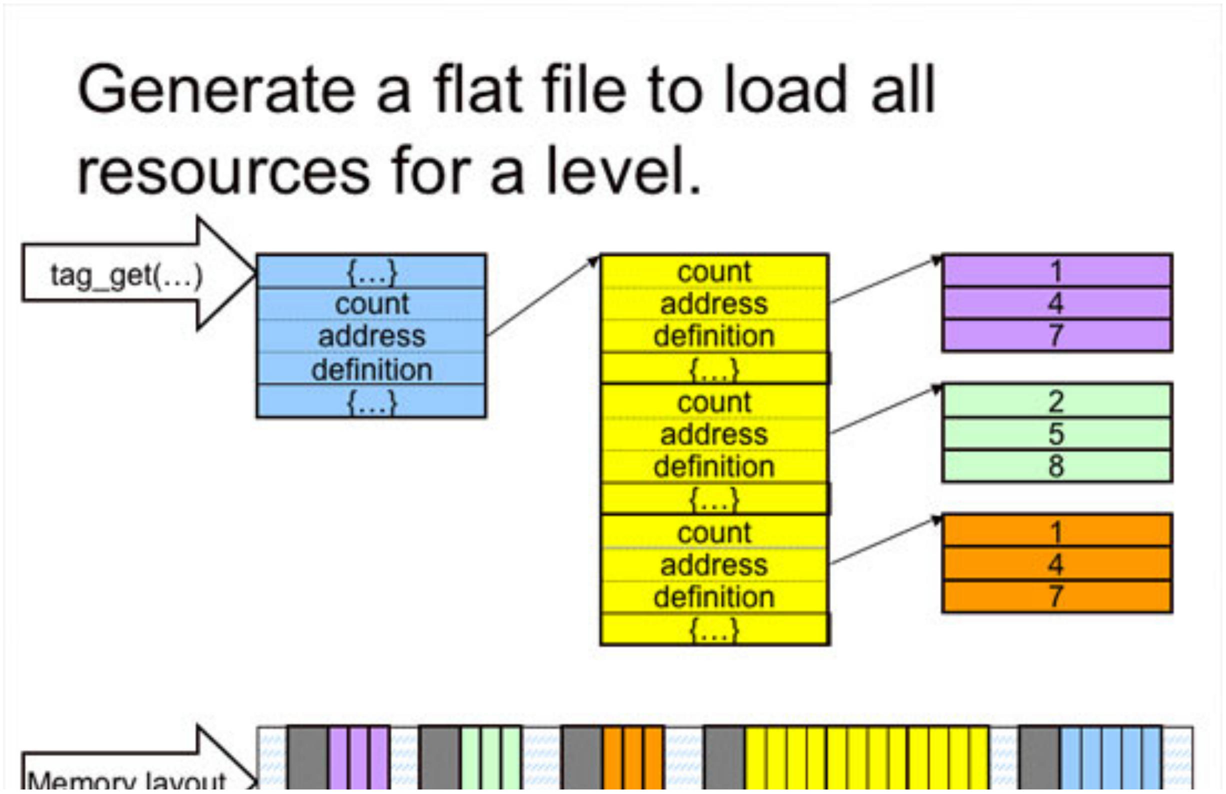
Optimize storage layout.





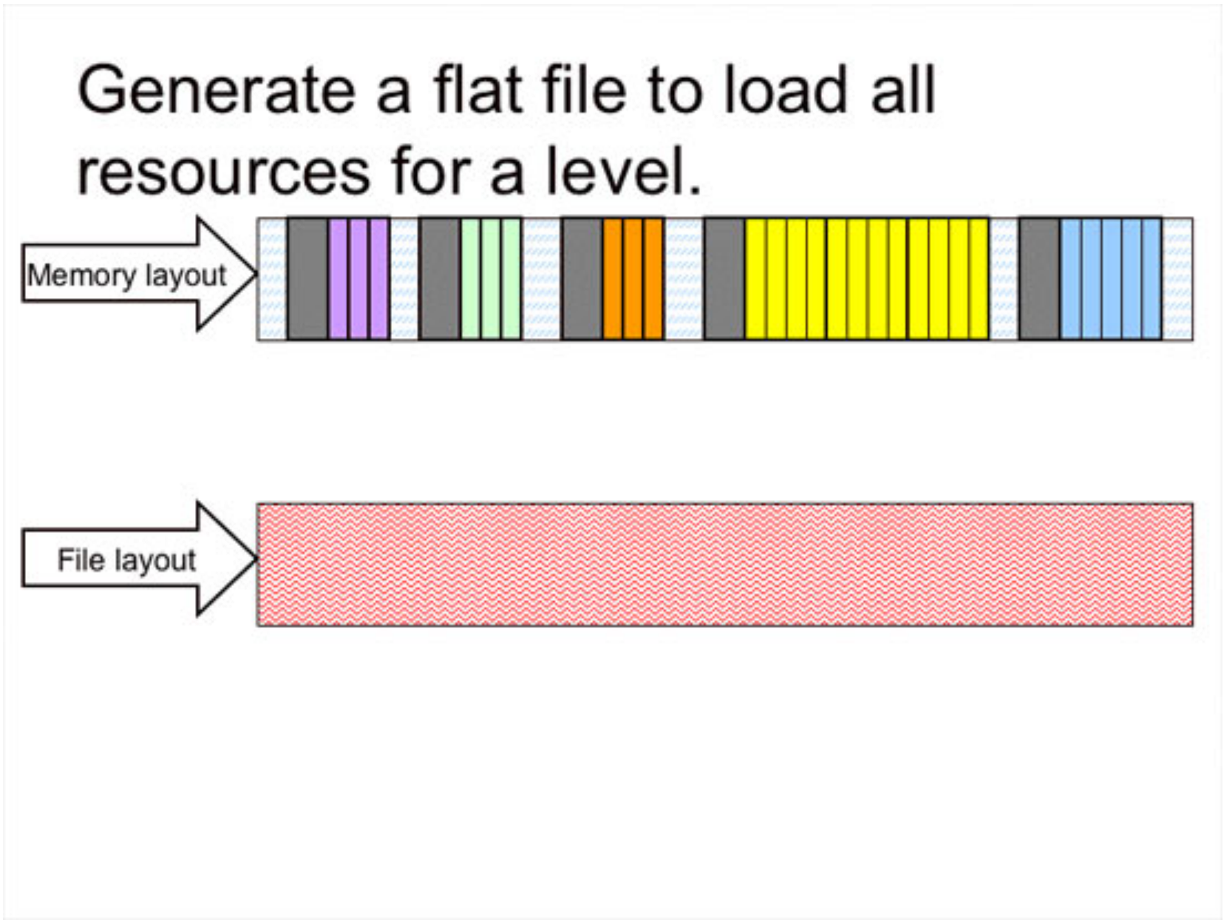
Clearly our tag system is a powerful resource management system for development builds. It is also a great system for optimizing content for our shipping build.

One powerful aspect of our tag system is the complete separation between resource access and resource storage at runtime. Accessing tag data must be done through the tag system; it can provide that data through any arbitrary mechanism: caching the data in memory, generating it at runtime, streaming it over the network, etc. As such, we can completely change the underlying implementation of the tag system without affecting any other game system that relies on the tag system. The fact that our editing build runs from multiple single files is an implementation detail; it is not a necessary component of the tag system itself.

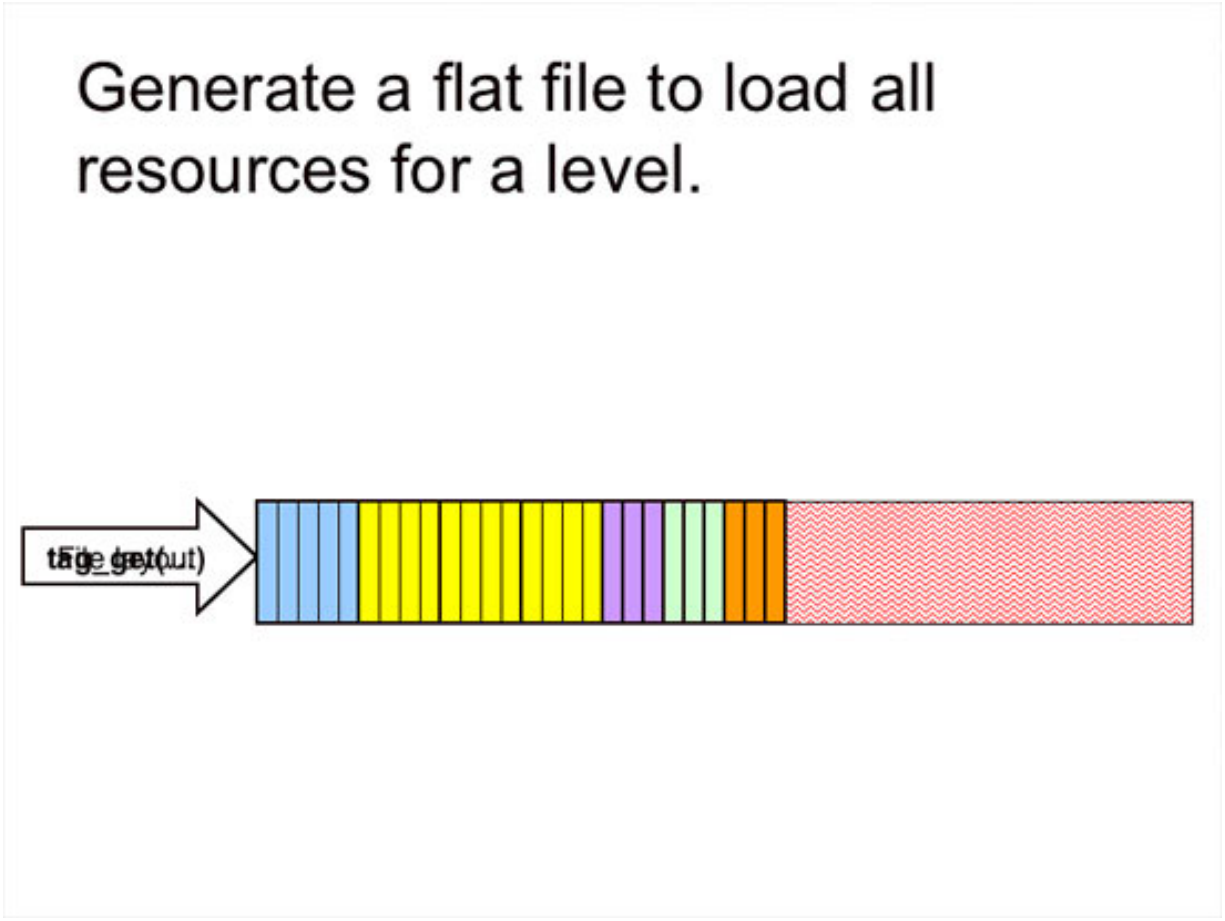


Generate a flat file to load all resources for a level.

Our single file based tag system is great for editing the game, but to run on retail Xboxes we need to optimize the resource format so that it loads quickly with a minimal amount of memory; loading from multiple single files would clearly have a lot of runtime overhead, as well as rebuilding tags from scratch every time we run a new level. Since all tags can be referenced globally by a handle, once we load a level we have all the assets we needed to load it again. And, because we have the schema for every loaded tag, we know how to serialize each tag to a fixed runtime address without having to write custom code. When we build the final resource file, which we call a **cache file**, we simply dump all loaded tags into a giant buffer and write that out to a single file. At runtime, we just need to read that file into a fixed address before we start the level; all other runtime systems behave the same. We do have a few custom steps for stripping out and storing demand loaded data (namely textures and sound), but the overall amount of work to implement the shipping resource files is extremely straightforward.



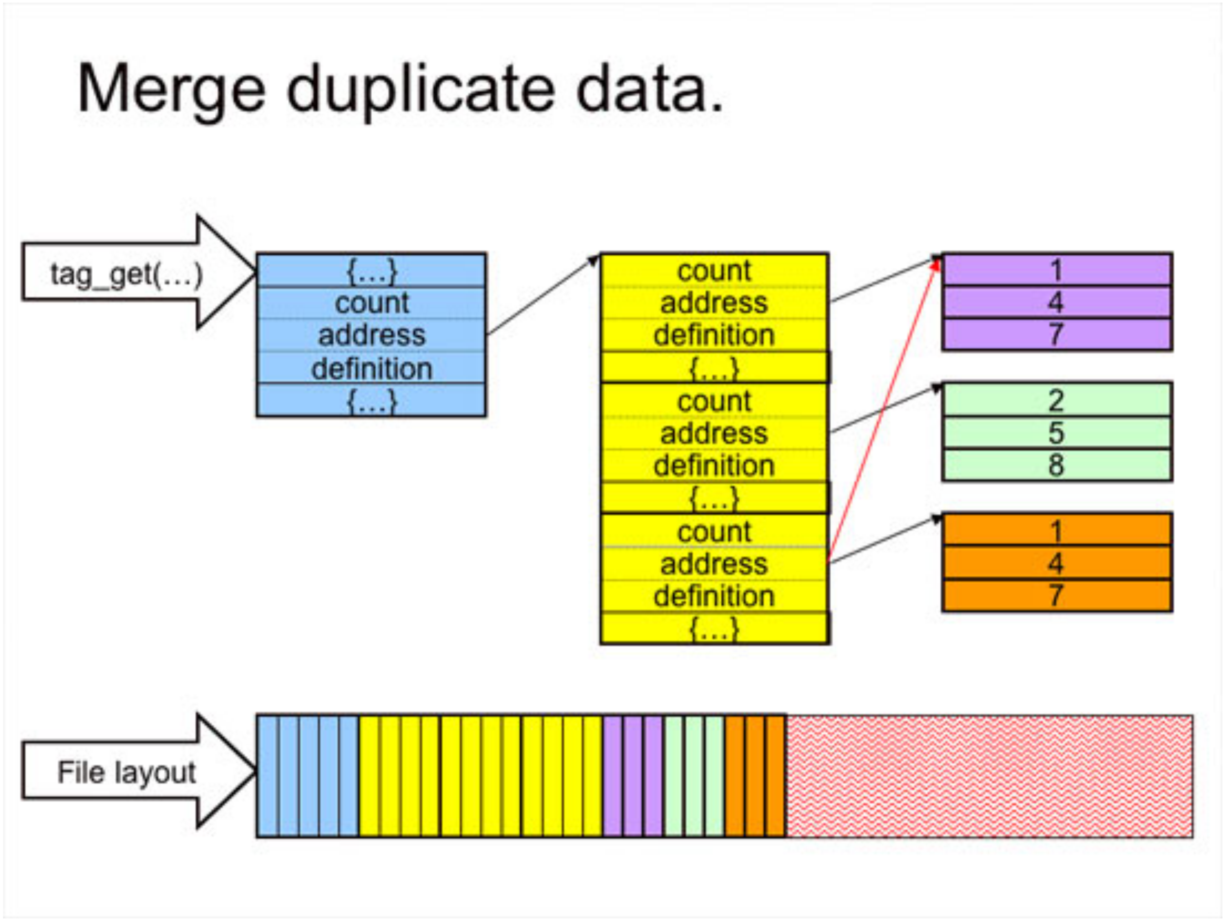
Our single file based tag system is great for editing the game, but to run on retail Xboxes we need to optimize the resource format so that it loads quickly with a minimal amount of memory; loading from multiple single files would clearly have a lot of runtime overhead, as well as rebuilding tags from scratch every time we run a new level. Since all tags can be referenced globally by a handle, once we load a level we have all the assets we needed to load it again. And, because we have the schema for every loaded tag, we know how to serialize each tag to a fixed runtime address without having to write custom code. When we build the final resource file, which we call a **cache file**, we simply dump all loaded tags into a giant buffer and write that out to a single file. At runtime, we just need to read that file into a fixed address before we start the level; all other runtime systems behave the same. We do have a few custom steps for stripping out and storing demand loaded data (namely textures and sound), but the overall amount of work to implement the shipping resource files is extremely straightforward.



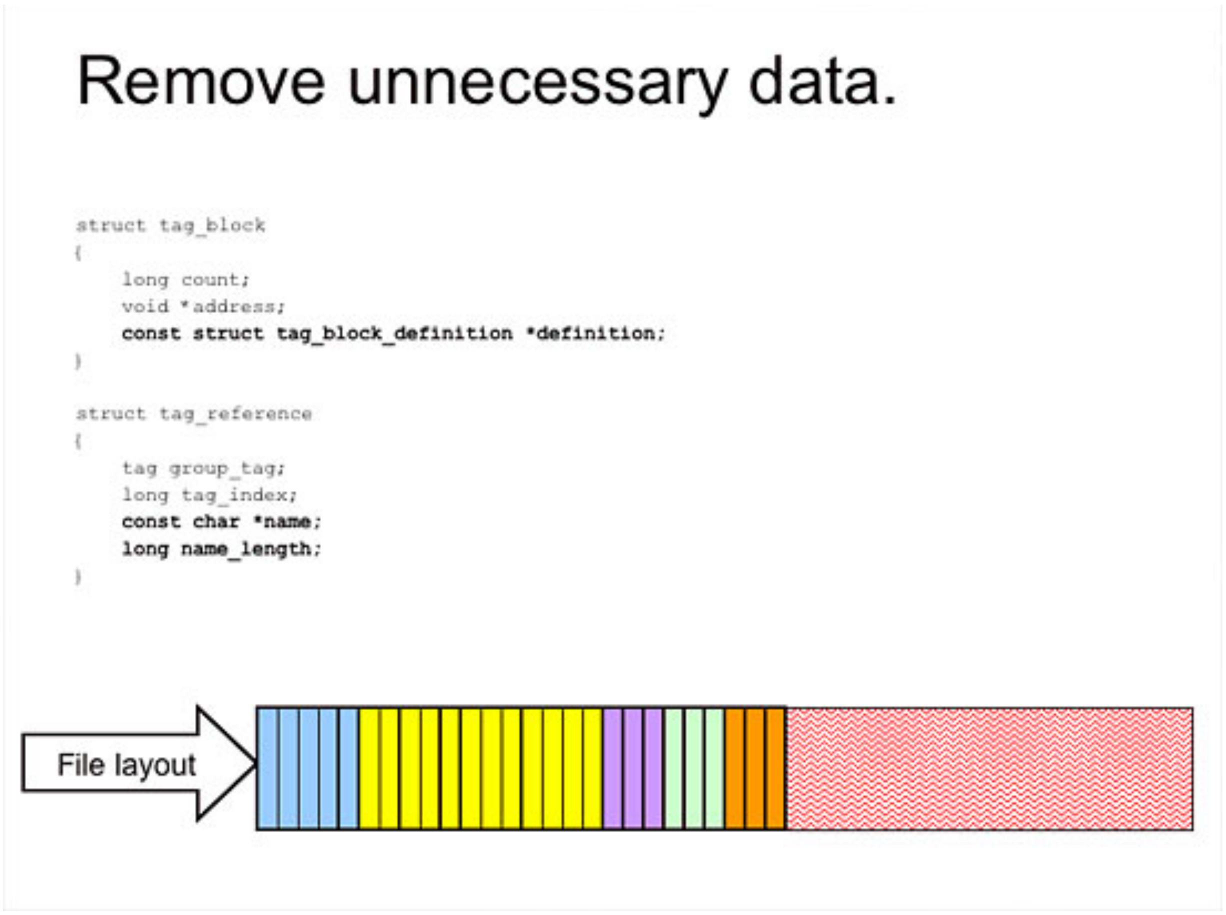
Generate a flat file to load all resources for a level.

Our single file based tag system is great for editing the game, but to run on retail Xboxes we need to optimize the resource format so that it loads quickly with a minimal amount of memory; loading from multiple single files would clearly have a lot of

amount of memory, loading from multiple single files would clearly have a lot of runtime overhead, as well as rebuilding tags from scratch every time we run a new level. Since all tags can be referenced globally by a handle, once we load a level we have all the assets we needed to load it again. And, because we have the schema for every loaded tag, we know how to serialize each tag to a fixed runtime address without having to write custom code. When we build the final resource file, which we call a **cache file**, we simply dump all loaded tags into a giant buffer and write that out to a single file. At runtime, we just need to read that file into a fixed address before we start the level; all other runtime systems behave the same. We do have a few custom steps for stripping out and storing demand loaded data (namely textures and sound), but the overall amount of work to implement the shipping resource files is extremely straightforward.



We try to keep game systems from modifying tag data. Since we don't need to worry about most systems modifying tag data, we can go ahead and merge duplicate tag memory when we are building the cache file. We do this by aliasing tag block elements when we detect duplicate binary data. When we shipped, this process netted us an additional 1 MB per level.

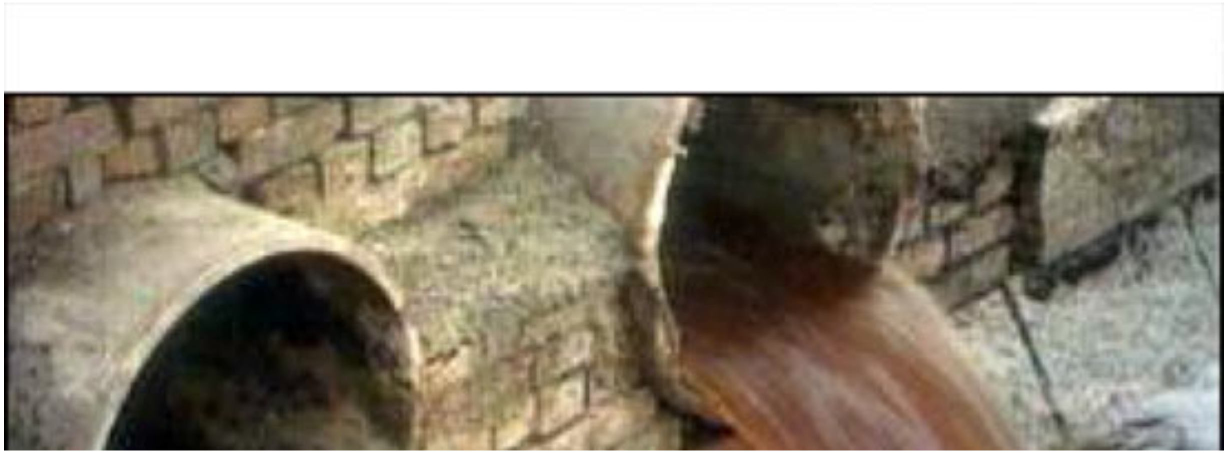


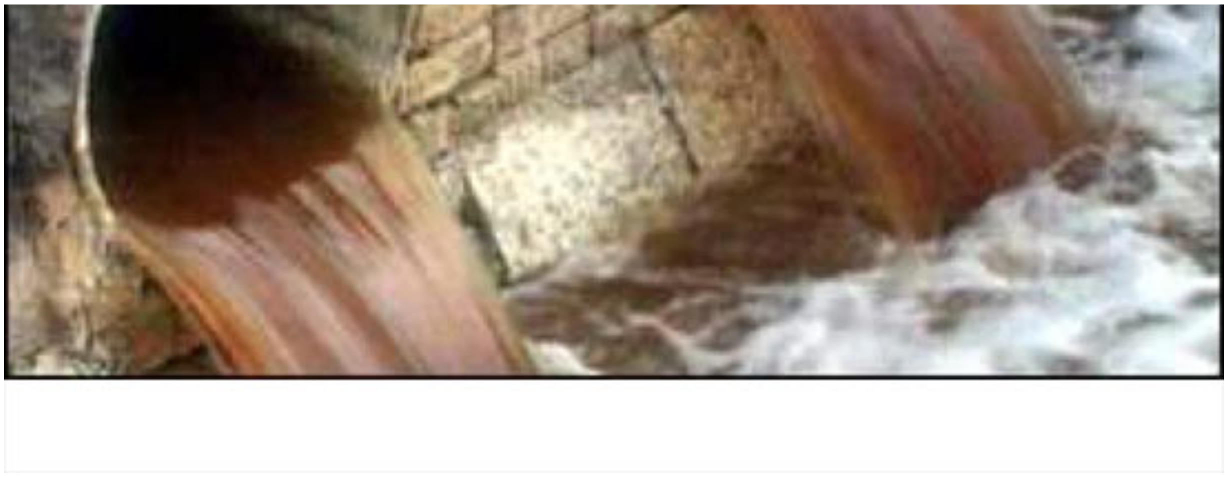
Remove unnecessary data.

```
struct tag_block
{
    long count;
    void *address;
    const struct tag_block_definition *definition;
}

struct tag_reference
{
    tag group_tag;
    long tag_index;
    const char *name;
    long name_length;
}
```

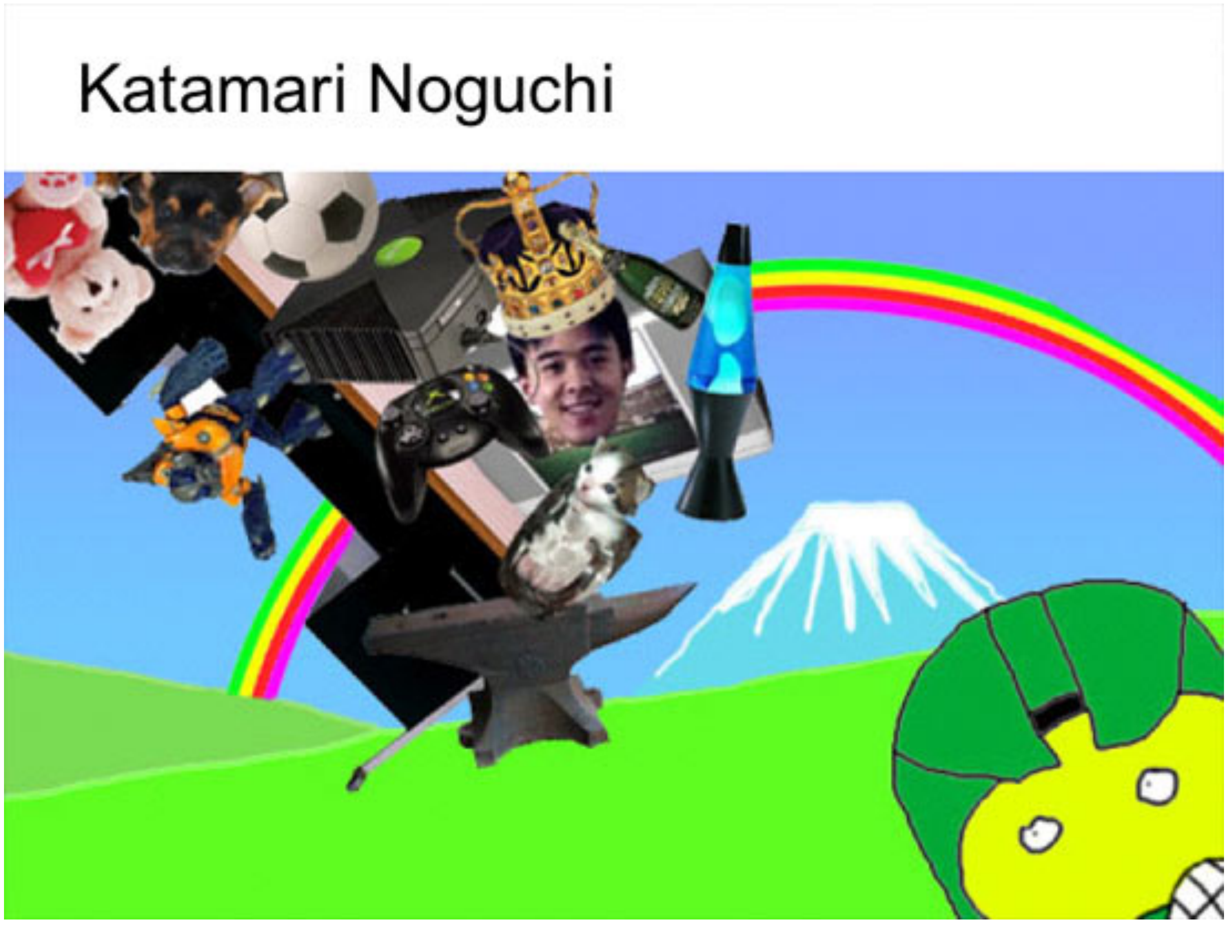
File layout

We also used the tag layout to further optimize the resource layout for our DVD build. Tag_references, tag_blocks, and tag_data contain members that are not necessary for our shipping build. As you can see from the declaration of a tag_block, there is a pointer to the tag_block's definition; this pointer is used to map type information to a tag block. Since tags cannot be modified in the shipping build, there is no need to keep this member around. We stripped this data when we built a cache file. For a typical Halo 2 level, this saved ~700k to 1MB.





Like everything else in game development, we did have a few issues with the tag system during Halo 2. One could easily classify the problems into two categories: problems of design, and problems of scale. I'm going to concentrate on scalability problems, mainly because those are more interesting.



One design problem that turned into a scalability problem late in development centered around versioning. Simply put, w\e didn't have a good versioning system for tags for Halo 2. This in and of itself would not have been a problem except for how we went about future proofing tags in Halo 1. In order to make sure we could add fields to tags, we would often insert lots of unused padding into tags as we created new ones to make sure we could co-opt that space later. This wasn't a big deal for Halo 1, but for Halo 2, this unused space started to add up as our tag counts exploded. On two separate occasions I had to go in and strip out the unused space in all the tags in our content tree. You can imagine this being a pretty painful process.

Combat dialog destroys everything

The combat dialogue we had for Halo 2 was one of the most horrific data sets we could have ever created for any game for three major reasons:

Increased complexity of the combat dialog system required an equal increase in quantity of combat dialogue sounds.

Spoken dialog had multiple language versions embedded in a single sound tag for localization purposes.

Raw sound data and tag data were interleaved throughout the sound tag.

These three factors caused the Windows file system to break when we added combat dialogue for all the major characters. Running a level now required us to read in a relatively small amount of data over thousands of files with a non-predictable access pattern. This kind of interaction with the file system managed to thrash the file system cache every time we loaded a level that had characters with combat dialogue, which inconveniently happened to be every single campaign level. This had the unfortunate side effect of exploding level loading times on the PC for everyone.

The best solution we came up with was to unmap combat dialogue for most of the artists. Which is to say we didn't really solve this problem.

Abstract

Creating an immersive gameplay environment requires more than utilizing the latest shader technology or implementing cutting edge-AI; it requires the content to drive it. As games become bigger and better, content management becomes more and more important. With 12 GB of just in-game assets spread across 39,000 files, we had to build custom tools to effectively handle content management for Halo 2. Even simple concepts like just-in-time editing of game assets or simply loading data from disk turn into complex beasts of implementation at that scale.

Introduction

As games become more and more complex and as hardware limitations have less and less of an impact on the quality of a game, future game innovation will come less from technology and more from content. For the purposes of this paper, content management is the process by which you create, organize and access non-volatile data necessary at runtime to drive a game engine. Content management applies to both runtime usage and persistent storage of game assets. For Halo 2, we were fortunate enough to have a mature persistent object system that handled all our resource management.

The tag system

A **tag** is the fundamental unit of our resource system. It corresponds to a single file on disk and a C structure in memory. Our tag system is similar to XML; where XML defines data as an XML file and its definition as a schema, we define data as a tag and its definition as a **tag_group**. However, there are two main differences: XML is stored as text while a tag is a stored as binary data, and an XML schema is defined in a text file while a tag_group is defined in code. We define tag_groups in code using a set of preprocessor macros that break down the corresponding C structure into its component types, e.g. integers, floats, strings.

Example 1: Defining a simple tag

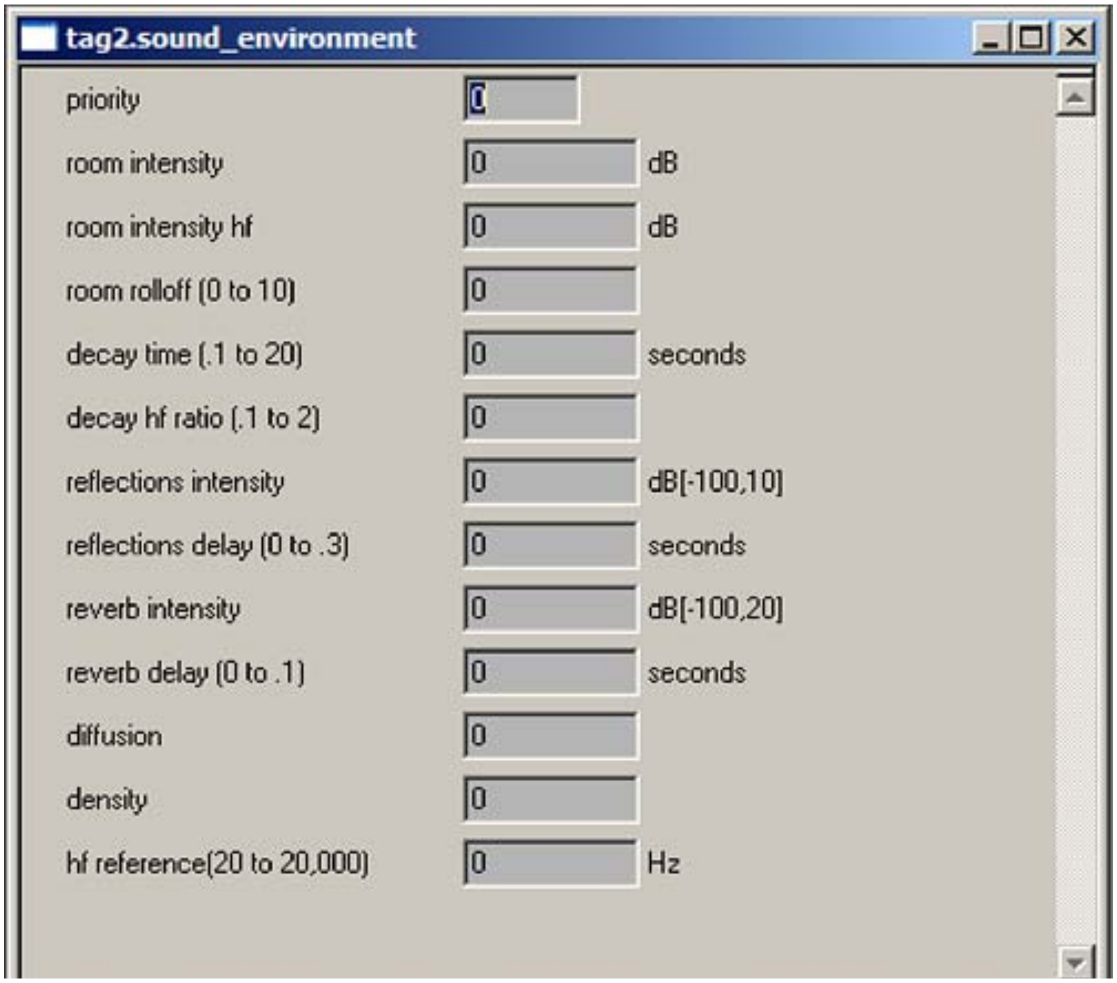
Given the following C structure:

```
typedef float real;


struct sound_environment
{
     real room_intensity_db;
     real room_intensity_hf_db;
     real room_rolloff_factor;
     real decay_time;
     real decay_hf_ratio;
     real reflections_intensity_db;
     real reflections_delay;
     real reverb_intensity_db;
     real reverb_delay;
     real diffusion;
     real density;
     real hf_reference;
};
```

We can define the corresponding tag_group in C code as follows:

```
const unsigned long SOUND_ENVIRONMENT_TAG= 'snde';

TAG_GROUP(
sound_environment,
SOUND_ENVIRONMENT_TAG
SOUND_ENVIRONMENT_TAG,
sizeof(sound_environment))
{
     {_field_real, "room intensity"},
     {_field_real, "room intensity hf"},
     {_field_real, "room rolloff (0 to 10)"},
     {_field_real, "decay time (.1 to 20)" },
     {_field_real, "decay hf ratio (.1 to 2)"},
     {_field_real, "reflections intensity:dB[-100,10]"},
     {_field_real, "reflections delay (0 to .3):seconds" },
     {_field_real, "reverb intensity:dB[-100,20]"},
     {_field_real, "reverb delay (0 to .1):seconds"},
     {_field_real, "diffusion"},
     {_field_real, "density"},
     {_field_real, "hf reference(20 to 20,000)},
};
```

The sound_environment tag_group can then be inspected at runtime; our game asset editor, Guerilla, uses this information to generate a simple editing interface:



priority	0	
room intensity	0	dB
room intensity hf	0	dB
room rolloff (0 to 10)	0	
decay time (.1 to 20)	0	seconds
decay hf ratio (.1 to 2)	0	
reflections intensity	0	dB[-100,10]
reflections delay (0 to .3)	0	seconds
reverb intensity	0	dB[-100,20]
reverb delay (0 to .1)	0	seconds
diffusion	0	
density	0	
hf reference(20 to 20,000)	0	Hz

To allow for more complex data structures than a simple C struct, we also have a field type called a **tag_block**. A tag_block is similar to a tag_group; it defines a resizeable array whose elements correspond to a C struct. Since both a tag_group and tag_block can contain any set of fields, including tag_blocks, we can describe almost any type of structured data hierarchy.

Example 2: Defining a more complex tag

C struct definition:

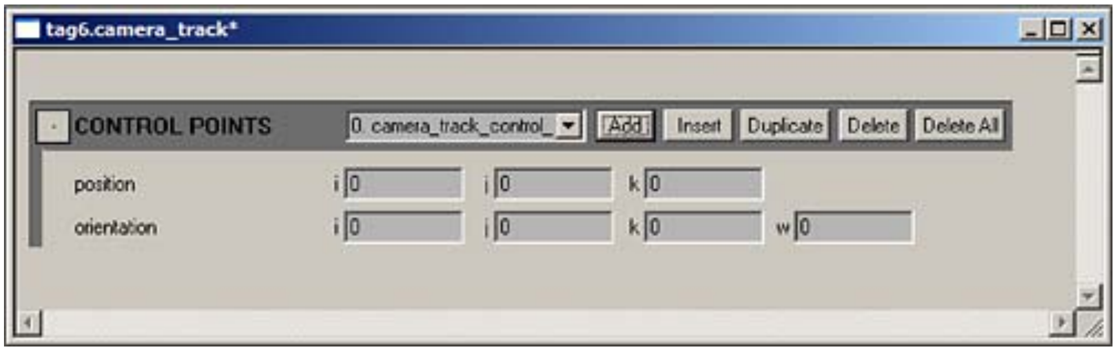
```
struct s_camera_track_control_point
{
     real_vector3d position;
     real_quaternion orientation;
};

struct s_camera_track_definition
{
     unsigned long flags;
     struct tag_block control_points;
};
```

Tag group definition:

```
const unsigned long CAMERA_TRACK_DEFINITION_TAG= 'trak';

TAG_BLOCK(
camera_track_control_point_block,
k_maximum_number_of_camera_track_control_points, sizeof
(s_camera_track_control_point))
{
     {_field_real_vector3d, "position"},
     {_field_real_quaternion, "orientation"},
     {_field_terminator}
};

TAG_GROUP(
camera_track,
CAMERA_TRACK_DEFINITION_TAG,
sizeof(s_camera_track_definition))
{
     {_field_block, "control points",
&camera_track_control_point_block},
     {_field_terminator}
};
```

};

Editing interface:



0 camera_track_control_

position i 0 j 0 k 0

orientation i 0 j 0 k 0 w 0

To access elements of a tag_block, you have to use a macro:

```
s_camera_track_definition *camera_track= ...;
long control_point_index= ...;
s_camera_track_control_point *control_point=
TAG_BLOCK_GET_ELEMENT(
&camera_track->control_points,
control_point_index,
s_camera_track_control_point);
```

This is an implementation detail that can easily be abstracted out via more preprocessor macros or through templated wrappers on top of the tag_block structure.

We also have a tag type to allow tags to cross-reference one another, called a tag_reference. A tag_reference is essentially a typed persistent pointer that is maintained by the tag system when a tag is loaded. A tag_reference can be further defined to point to tags of a single tag_group or multiple tag_groups. This simple abstraction is what powers much of our game asset management. For example, if a UI element needs a texture to render correctly, its tag_group would have a tag_reference reference field to a bitmap tag instead of embedding a bitmap directly.

Example 3: Defining a tag that depends on other tags

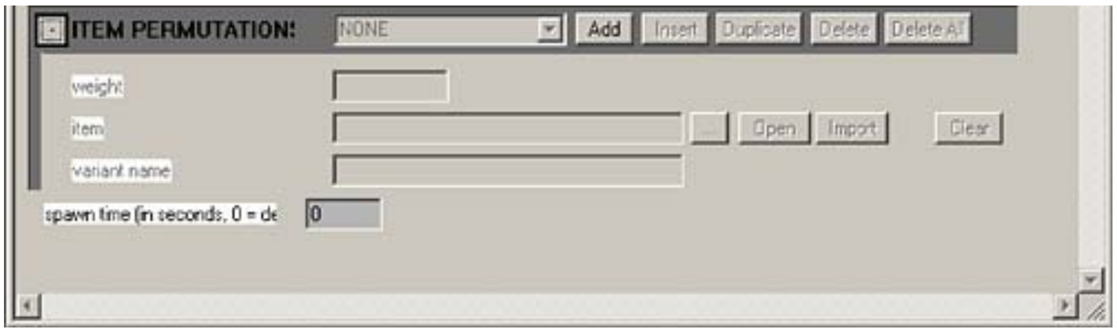
C struct:

```
struct item_permutation_definition
{
     real weight;
     struct tag_reference item;
     string_id variant_name;
};

struct item_collection_definition
{
     struct tag_block permutations;
     short spawn_time;
     short pad;
};
```

Tag group definition:

```
extern const unsigned long ITEM_DEFINITION_TAG;

TAG_REFERENCE_DEFINITION(
global_item_reference,
ITEM_DEFINITION_TAG);

#define MAXIMUM_NUMBER_OF_PERMUTATIONS_PER_ITEM_GROUP 32
#define MAXIMUM_NUMBER_OF_ITEM_GROUPS 128

TAG_BLOCK(
item_permutation,
MAXIMUM_NUMBER_OF_PERMUTATIONS_PER_ITEM_GROUP, sizeof(struct
item_permutation_definition))
{
     {_field_real, "weight "},
     {_field_tag_reference, "item ", &global_item_reference},
     {_field_string_id, "variant name"},
     {_field_terminator}

};

TAG_GROUP(
item_collection,
ITEM_COLLECTION_DEFINITION_TAG,
sizeof(struct item_collection_definition))
{
     {_field_block, "item permutations", &item_permutation},
     {_field_short_integer, "spawn time (in seconds, 0 =
default)"},
     FIELD_PAD(1 * sizeof(short)),

     {_field_terminator}
};
```

Editing interface:



Both tag_blocks and tag_groups can also have load-time processing behavior defined in code, e.g., to optimize data structures or initialize runtime variables. We refer to this load-time processing as postprocessing. For example, our sound effect tag generates different C structs at runtime depending on the type of sound effect; a distortion effect would take the high-level distortion parameters to create the DirectSound distortion parameters and store that data in the tag. Our shader system has a similar mechanism to take a high level shader definition and optimize it into a more code and hardware friendly format at runtime.

All this manual maintenance of the tag system may seem like a lot of tedious work, but it gives us the benefits one would normally get with a native reflection system. We can write code that analyzes or transforms any tag in a generic fashion. Many operations that can be defined at a high level using type-agnostic pseudocode can be implemented in about the same amount of C/C++ code that manipulates the tag system.

One obvious application of the tag system is the ability to serialize a tag from memory to disk as well as restore a tag in memory from the disk without having to write custom code for each particular asset type; loading a tag is a fairly straightforward process:

1. Find the tag_group corresponding to the tag we wish to load
2. Load the tag into memory
3. Walk the tag hierarchy and postprocess each tag_block element using a post-order traversal
4. Walk the tag hierarchy again and for each tag_reference find or load the corresponding tag (starting at step 1)
5. Postprocess the tag using the postprocess function defined its tag_group

This load process ensures that once a tag is loaded, it has every tag it depends on also loaded. This process also allows for tags that are multiply referenced to be loaded once at runtime. We can also provide default tags for tag_references that point to non-existent tags. For example, we have a default object tag that is a horribly textured sphere; should any tag reference a non-existent object tag, we show the ball instead.

The payoff[s]

Our tags system as a persistent object system helped us tremendously in the development of Halo 1 and 2.

Level loading process

With most of our game assets defined as tags, none of our leaf game systems (e.g., physics, sound, rendering, AI) manage their respective assets; all resources are handled by the tag system. To facilitate this separation of control between the game and the tag system, we only explicitly load two tags: the **globals** tag and a **scenario** tag. The scenario tag is basically the level representation. The globals and scenario tags are available for any game system to access, so for a game system to have access to a particular tag, it must be accessible through the globals tag or the scenario tag.

This consistent loading behavior across the entire game provided us a single entry point for loading a level for gameplay, lightmapping, optimizing for a DVD build, or any other process we wish to apply to an entire level. However, because the only supported way to run the game requires a level to load, it has been very difficult for us to develop interactive tools for viewing and editing data that doesn't need a level loaded, e.g., interactively editing particle effects or viewing animations. What usually happened is that we added scripting commands in the game engine to manually load a given tag and use the automatic reloads to update it on the fly.

Automatic reloading of single tags

Since we only allow access to tag data through handles provided by the tag system, a tag can be reloaded or modified at any time during the lifetime of a game quickly and easily, without adversely affecting other game systems. To make this system more robust, we have a single entry point for reloading tags in case we have a need to have custom cleanup or initialization code when a tag reloads. For example, we pushed data from tags into Havok, a third party physics system. Any time we reloaded a physics tag we had to release all references Havok had on the tag before unloading it from memory, and re-associate those references once the tag was fully reloaded.

For our PC builds, we can easily detect file changes using the file system

For our PC builds, we can easily detect file changes using the file system change notification system. For our Xbox builds, the process is a little more complicated. Our Xbox build cannot use the same file paths as our PC build for performance reasons related to the Xbox's simplified file system. In order to accommodate this restriction, we encode tag paths into a number and generate a directory and filename based on that number. We store this information for all tags on the Xbox in a giant file called the tag file index and use that mapping at runtime to locate tags based on their full path. We use that information every time we synchronize tags from the PC to the Xbox so that we only need to copy the tags that have changed. We call this synchronization process **xsync**. If the game is running on the Xbox during an xsync, it can detect when the index file updates and reload those tags that have changed in the process. Xsync is an integral part of our content pipeline; artists and designers can changes made on the PC quickly on the Xbox, allowing for quick iteration times on content presented as close to the final presentation as possible. For example, an artist can update the textures on a model on the Xbox as s/he edits them on the PC; for them, the average turnaround time to see a texture update is about 5 seconds.

Cache files: Optimized resources for the Xbox

One powerful aspect of our tag system is the complete separation between resource runtime access and resource storage. Accessing tag data must be done through the tag system; as such, the tag system can provide that data through any arbitrary mechanism: caching the data in memory, generating it at runtime, streaming it over the network, etc. As such, we can completely change the underlying implementation of the tag system without affecting any other game system that relies on the tag system. The fact that our editing build runs from multiple single files is an implementation detail; it is not a necessary component of the tag system itself.

Our single file based tag system is great for editing the game, but to run on retail Xboxes we need to optimize the resource format so that it loads quickly with a minimal amount of memory; loading from multiple single files would clearly have a lot of runtime overhead, as well as rebuilding tags from scratch every time we run a new level. Since all tags can be referenced globally by a handle, once we load a level we have all the assets we needed to load it again. And, because we have the tag_group layout for every loaded tag, we know how to serialize each tag to a fixed runtime address without having to write custom code. When we build the final resource file, which we call a **cache file**, we simply dump all loaded tags into a giant buffer and write that out to a single file. At runtime, we just need to read that file into a fixed address before we start the level; all other runtime systems behave the same. We do have a few custom steps for stripping out and storing demand loaded data (namely textures and sound), but the overall amount of work to implement the shipping resource files is extremely straightforward.

Geometry cache (AKA, the "anything" cache)

For Halo 2, we decided to add geometry as a cacheable data type. This wasn't nearly as easy to do compared to texture or sound data because we had two different geometry formats, both of which were more complex than just a chunk of data. In order to correctly page in this kind of data, we put the geometry data into a tag_block and inspected its structure to serialize it similar to how we serialize cache files. At runtime, we would inspect the tag_block structure to restore the data properly. This kind of caching mechanism for geometry allowed us to put much more variety of models and environments into every level. It wasn't a perfect caching mechanism, as we did end up with many levels that had too much data to put into memory at once. But in the end, it gave us the flexibility to build much more detailed and diverse worlds compared to Halo 1.

Incidentally, because the geometry cache operated on single-element tag_blocks and not just geometry specific data, we were also able to cache our lip-sync data for combat dialogue using the same system.

Tag dependency database

During the course of Halo 2's development, we found the need to analyze the relationships between tags, e.g., locating dangling tag references, finding what tags reference a given tag, etc. In order to analyze these relationships, we loaded every tag and stored all its tag_references. After doing this for every tag, we would then go back and create a graph with a link for each tag to the tags that reference it.

This dependency database also gave us enough information to move entire tag hierarchies around without breaking external references; given a set of tags and the location to move them to, we could determine what tags needed to have their references fixed up and move the tags safely without breaking dependencies. We also used the same information to clone entire tag hierarchies; this let us take existing content and replicate it into its own self-contained hierarchy for further modification without affecting the pre-existing content.

These are a few of the many powerful analysis tools available once you have a type inspection system that is programmatically accessible for your game assets.

Problems with the tag system

Unfortunately, we ran into several problems with our tag system over the lifetime of Halo's development.

Data coupling require more complex reload behavior

As we evolved existing systems, we had tags depend on data in their child tags at runtime, e.g., a sound effect would depend on data in a sound_effect_template tag; changing the child tag would have to trigger a reload of the parent as well in order to maintain a valid runtime state. This was easily solved by generating a mapping from child tags to parent tags; the hard part was determining which child<->parent tag relationships to map. Mapping the dependencies based solely on the tag_group layout wasn't an option for two reasons: many tags did not have their runtime behavior defined from data in child tags, or they had behavior defined by data in a child tag of a child tag. Our solution was a bit more elegant: every time we postprocess a tag, we log which tags are accessed via the global handle system, and store that as the dependency. At tag reload time, we took the list of tags to reload, added the tags that depended on them at runtime, and then reloaded them in the appropriate order.

Unfortunately, the code-driven dependency database caused some strange runtime behavior. Many tags were validated at runtime during the postprocessing of a more global tag. In one particularly horrible case, the shaders attached to our level geometry were validated at postprocess time for level geometry. This had the unfortunate side effect of reloading the level geometry when a bitmap attached to a shader attached to the level geometry changed. Reloading level geometry takes a while because of how it globally affects the game, so even simple bitmap changes could take a few minutes to reload. In order to break these kinds of unnecessary dependencies, we had to manually suppress the dependency mapping process in the postprocess function as they were discovered. As you can expect, this manual process was extremely brittle, and it caused a lot of frustration as the code evolved and created new unexpected dependencies between tags.

More complex tags break Windows

Our basic game asset editor, Guerilla, is a powerful editing tool that uses the

Our basic game asset editor, Guerilla, is a powerful editing tool that uses the tag_group layout to create a simple editing interface for tags. Guerilla iterates over the tag_group fields and generates the editing interface by compositing pre-defined dialogs together to create a form view. Unfortunately, creating dialogs allocates space out of the Windows desktop heap. Since every GUI application uses the desktop heap, exhausting it will cause problems with every GUI application. As our tags became bigger and more complex, Guerilla would often exhaust the desktop heap. This usually manifested in broken behavior across all applications: error dialogs would not display, chat programs wouldn't run correctly, 3d Studio Max would mysteriously eat input without displaying the correct editing dialogs. Our solution, which is more accurately described as a kludge, was to hide fields that only programmers needed to view. This wasn't really a solution; it merely delayed Guerilla's ultimate destruction of Windows.

Tag system destroys the Xbox file system

For our Xbox editing build to run with the new geometry cache system, we had to dump the stripped geometry into a giant monolithic file. Our first pass at this process had us strip the geometry and other cacheable data at load time on the Xbox. At first, this solution worked pretty well. Cacheable data was stripped at load time, resulting in a smaller memory footprint, and our editing build only had to access a single file to page in cacheable data. However, this resulted in an explosion of load times later on in the project as we added a ton of sound assets. Sound assets by far dwarf every other kind of asset we have, in terms of variety and sheer size. We spent more time writing out cacheable sound data than we did reading in tags. To reduce load times, we moved the cacheable data stripping into the xsync process. This increased xsync times a lot, but it reduced Xbox load times enough to make it a net win in terms of overall time wasted waiting to run the game. We did run into a few runtime consistency problems, as the xsync process wasn't completely transactional. You could occasionally have a tag loaded at runtime that failed to cache its data properly at a later xsync time making the tag basically have no valid cacheable data. However, this was due to space constraints on the Xbox and is something that is easily solvable in the future.

Combat dialogue destroys the Windows file system

The combat dialogue data we had for Halo 2 was one of the most horrific data sets we could have ever created for any game for three major reasons:

1. The order of magnitude increase in complexity with our combat dialog

system required an equal increase in quantity of combat dialogue sounds.

2. To ease our localization process, spoken dialog had multiple language versions embedded in a single sound tag.
3. Raw sound data and tag data were interleaved throughout the sound tag.

These three factors caused the Windows file system to break when we added combat dialogue for all the major characters. Running a level now required us to read in a relatively small amount of data over thousands of files with a non-predictable access pattern. This kind of interaction with the file system managed to thrash the file system cache every time we loaded a level that had characters with combat dialogue, which inconveniently happened to be every single campaign level.

Thrashing the Windows file system cache is one of the most effective ways to de-optimize any file system activity; any file system data that could be read from memory now has to come from the disk, which is obviously an order of magnitude slower. For us, that caused load times to increase from 15-20 seconds to 5-8 minutes for the more complex campaign levels, even on machines with 2 GB of RAM.

To maintain reasonable load times on the PC, we resorted to shadowing the combat dialogue directory from source control. Obviously, this wasn't the most optimal solution, but short of writing our own PC side file system, it was the best solution we could come up with.

Future considerations

As powerful as our tag system is, it is still cumbersome to setup new tag_groups and tag_blocks. A programmer has to manually define a tag in code, a process that is both extremely error prone with those unfamiliar with the system and quite brittle as data formats change and evolve. A more maintainable solution would be to separate the tag_group definition into a separate file and build process; one major benefit of a separate system would be to provide for a less brittle process for defining a tag_group and making sure the layout is consistent with the compiler's struct layout.

Another area rife with possibilities would be to allow for arbitrary data stores to serve as the backend of the tag system. As it stands right now, the tag storage system is basically a database implemented on top of a file system. We can easily abstract out the storage system so that we can load tags from anywhere; a network share, a database that generates the tag on the fly, from a collection of text files, or any other mechanism we come up with.

Conclusion

Our tag system was an integral part of our overall development process that enabled us to effectively develop many different systems for Halo 1 and 2 while at the same time providing a resource management system that could be targeted to widely varying runtime platforms without affecting other code. By creating a cleanly separable persistence system that programmatically provides type inspection, we were able to analyze and optimize our resources quickly and easily with a minimum amount of effort and with very little modification of the underlying system.