

Dealing With Large 3D Worlds By Greg Corson

Office Contact
NEC Electronics Inc.
9005 Mountain Ridge Drive
Suite 240
Austin, TX 78759
(512) 502-2917
greg_corson@el.nec.com

Home Contact
12345 Alameda Trace Circle
Apt 313
Austin, TX 78727
milo@austin.rr.com

Abstract

Most games today break their worlds into pieces or levels based on limitations imposed by texture or system RAM. This paper discusses a number of techniques that can be used to make a single seamless world of considerable size. These techniques use prediction methods as well as systems similar to portals or “potentially visible sets” to determine in advance what 3D data is going to be needed and arrange to have it loaded before the player gets there. The same techniques are used to decide when to unload data that needs to be written back to disk because the player changed the world somehow.

Introduction

Most 3d games today have much more data and a much bigger 3D world than will fit into even a high-end PC. In the past most designers dealt with this problem by breaking their games up into “levels”, areas that would completely fit in system/texture RAM on their minimum target system. Then when the player moves from one level to another they would unload the data for the previous level and load the data for the next one.

Unfortunately, this technique causes a lot of design and playability problems. The process of loading a new level takes a noticeable amount of time and breaks up the flow of the game with “loading please wait” messages. This makes the game less immersive and less fun to play, particularly if the breaks are frequent. It also causes game design problems because levels have to be created to fit in this space budget and need to be connected in a limited number of places so loading messages don’t constantly interrupt the gameplay.

Breaking the game up into pieces this way also makes poor use of the hardware in many cases. Someone with 128M of RAM will have to endure just as many “loading please wait” pauses as someone with 32M, because regardless of RAM space the game still loads one fixed size level at a time. Picking a texture budget that will fit on any 3D card is equally difficult. In today’s market you are likely to see cards that can handle as little as 2M of textures to as much as 200M (hardware compressed). Clearly, we need to be more clever in our management of texture space, RAM and the computer in general so that our games will take better advantage of the hardware that’s there,

running acceptably on older/smaller computers while taking full advantage of the “Power Gamer” systems out there. The technique of breaking things up into levels doesn’t offer this flexibility now that computers and graphics cards cover such a wide range of performance.

This paper presents a variety of techniques that can be used to manage large 3D worlds in games. They come from a number of computer disciplines including graphics, database management and operating system design. To the best of my knowledge, none of the techniques are patented and have been in use for many years in mainstream graphics, but not in games. Proper use of these techniques should benefit both low-end and high-end target systems by making better use of what resources they have. They can also make game design easier because you don’t have to subdivide your game into small “level sized” pieces. The key techniques include:

- **Resource management** – You need to be able to efficiently manage all your game’s resources, like geometry, textures, scripts and so on. This includes knowing what resources are related to each other (like a model and it’s textures) so they can be managed together. The management scheme needs to be able to deal with resources spread over disk, RAM and texture RAM.
- **Resource Preparation** – In order to manage your game resources well and quickly, they need to be well indexed, with as much preparation as possible done before the game starts. Some of this preparation you can do during development, other bits can be done as the game starts up.
- **Prediction** – Your game must be able to predict in advance (a second or more) when it will need a resource like a texture. The further in advance you can predict these needs, the easier it will be to get the resources loaded in time without slowing down the game.
- **Profiling & Time management** – All these techniques require the game to do certain low priority tasks “in it’s spare time” such as when you are waiting for a page flip. For them to work you need to manage how your game spends it’s time so you always know how much spare time you will have between frames to do other things. You also need at least a minimal profile of the computer so you know what you have to work with in the way of speed, RAM and so on.
- **Cheap tricks** – Some well worn techniques to get you over the rough spots.

The difficulties of implementing these techniques in a game varies widely, but all of them deal with various forms of advance preparation, prediction and loading/saving data in the background while the game continues to run. Done correctly, they may even allow you to transparently page data from a CD that you would normally have made the user copy to the hard disk.

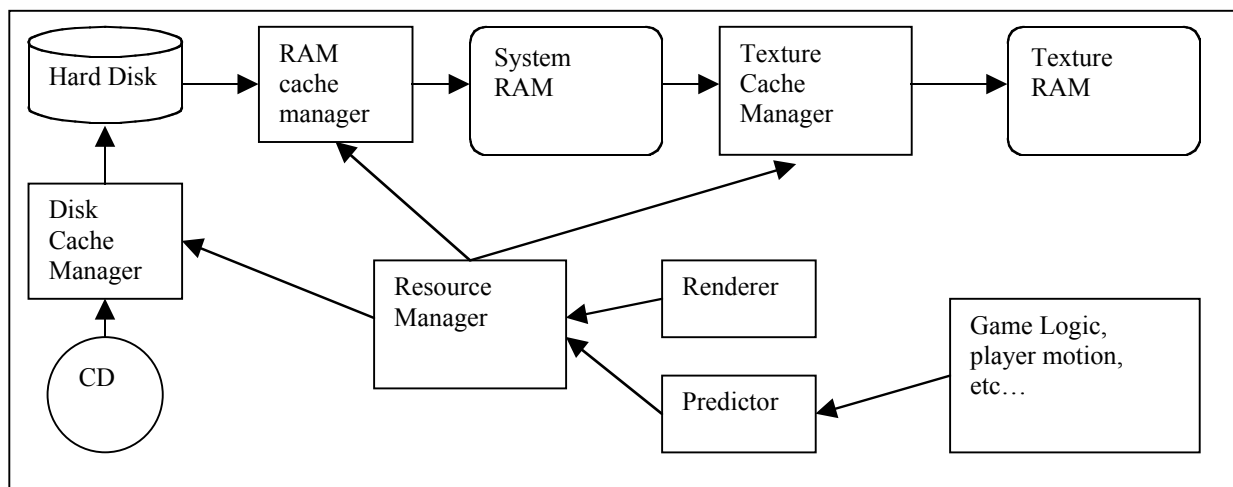
A Sample Resource Management Pipeline

Most programmers are familiar with the idea of a texture cache. This is typically an area in system RAM where all the textures for a game level are stored. Some cache

management software of what textures are loaded on the 3D card and if a texture is needed that isn't there, it transfers it up from RAM. When the 3D card runs out of space, textures that aren't being used are discarded from its RAM to make room for new ones.

The resource management system I'm proposing is very similar to one of these texture caches but it adds more layers of caching. It also introduces the idea of predicting the need for resources, such as textures, in advance so they can be loaded at a low priority that won't cause frame-rate stutters in the game.

The figure below illustrates the basic pieces of this resource management pipeline. Most resources (like geometry) would work their way up this pipe to system RAM and then be used. Some resources, like textures or sound sample banks, might have to go up one more level into a sound or 3D card's local RAM. In this case, I've shown the path all the way up to the texture RAM.



- **Disk Cache Manager** – This piece of code handles tracking the block of hard disk space we are using to cache data from the CD ROM. It will have a cache index that has an entry for every resource being cached on disk, followed by the location of that resource in the disk cache file. There will also be some information on when this item was last used, and some flag bits. The disk cache manager will also maintain a priority queue of items that are waiting to be loaded into the disk cache from the CD and it will have code to load them one at a time in a way that won't slow down the rest of the program. This would be done with asynchronous disk reads, a low priority background thread or some similar mechanism.
- **RAM Cache Manager** – This is basically the same as the disk cache manager, but it tracks the block of system RAM where we are caching data that came out of the disk cache. The only real differences is that it tracks RAM addresses instead of disk addresses.
- **Texture Cache Manager** – Again, much the same as the RAM Cache Manager but it's tracking the contents of texture RAM. The mechanism for loading data into texture RAM will have the same kind of queuing system as the first two caches, but

the transfer of data will be from RAM to RAM one texture at a time, or in short block copies.

- **Predictor** – This block of code is responsible for predicting resource needs in advance. Typically it will monitor the player's motion through the world and make guesses about where he might be in the next second or two. Then the resources for those areas are looked up and queued with the various cache managers so they can be loaded before the player gets there.
- **Renderer** – This is your renderer, shown because it may make specific high priority requests to the resource manager and caches. This would happen when, for example, prediction failed and the renderer discovered a texture it needed for the current frame wasn't there. The renderer would then make a crash priority request to have that texture loaded immediately. This would probably cause a frame rate glitch of some kind, but if the predictor is doing its job, this should never happen.
- **Resource Manager** – The resource manager is the overall traffic cop of the system. It monitors operation of the various caches and handles the master index of where the original copy of each resource resides. It needs to manage how much time the resource loading system is using and prevent the system as a whole from using so much time that it slows down the game. It will also handle staging of special types of requests such as "gotta have it NOW" requests that need to be filled as fast as possible or purge requests that allow the game to flush unused items out of the caches when required.

Cache Operation

Typically, as you go up from the CD through the various caches, the size of each cache will get smaller. For example, the CD might have 600meg of game data, the cache on hard disk around 200meg, the cache in system RAM 8 to probably no more than 64 meg (depending on RAM in the system) and the cached data in texture RAM somewhere from 4-14 meg. To make sure important data is always available as quickly as possible, you will want to make sure each level of the cache is as completely filled with relevant resources as possible.

When a game starts up, the resource manager and the predictor will take a look at the player's position and start requesting resources to be loaded into various levels of the cache. By the time the game starts, the top level caches will be will be loaded with the predictor's estimate of what will be needed in the next few seconds, the next level down will be bigger and have resources that are a bit farther out, and so on.

As the player moves through the world the predictor will search outwards from the player in the general direction he's moving for resources that are not loaded but likely to be needed soon. The predictor and resource manager will then make prioritized requests to the caches to load this data. Depending on how long the predictor thinks it will be before the resources are needed, the resource manager may only load a resource into the lower cache levels. On some occasions, you may find a texture, some geometry or other resource you need to render the current frame isn't loaded. In this

case, you will have to send a request to the caching system that will load this resource immediately, without going through the usual priority queues.

Also, keep in mind when dealing with texture data that, due to compression, the 3D card may actually be able to hold 200mb or more of texture data, more than will fit in system RAM. You need to make sure your cache code can deal with this case. If you don't handle it correctly, the large volume of textures passing through the RAM cache on the way to texture RAM could cause textures to be purged from system RAM before they have the chance to be loaded into texture RAM.

Building a Resource Cache

As you know, a Cache is simply a system that takes the most frequently used data found in some lower speed storage and makes copies of it in faster storage so when you need it you can get at it more quickly. Most of us have some familiarity with the primary and secondary caches that CPUs use to speed access to frequently used areas of system RAM. These types of caches tend to be tough to build and tune because they cache small blocks of a large amount of system RAM. For our purposes we are creating "resource caches" which are much simpler. A resource cache takes a whole resource such as a texture and treats it as one piece.

A cache needs an index, this is normally a structure which I refer to as a "cache line". Each cache line has a pointer to the location in the cache where a resource is temporarily stored. It will have some flags and some kind of age data that can be used to determine how long it's been since this resource was last used.

For caching to work you also need a "key" to identify the resource, the best way to do this is to have a tool automatically assign each resource a number (preferably consecutive) when the game is being built. These numbers can then be used as an index into an array of cache lines, or as a "key" that you can search on. The cache will also need some kind of storage allocator to parcel out space in the cache to each resource. When you're caching things in RAM, you can use something like malloc to do this. If you're managing a cache that's on a disk, you will need to write an allocator yourself. However you do it, you'll need to make sure that the allocator is smart enough so the cache doesn't become fragmented over time.

Finally, the cache needs to have a load queue. This queue is a list of items that have been requested to be loaded into the cache, in priority order. Whenever there's free time, the cache will try to load the highest priority item in the queue. Note that whenever an item is loaded into the cache, if there isn't room for it the cache code will purge some resources that haven't been used in awhile. The queue needs to be able to handle requests that will be loaded in the background at low priority, and requests that need to be loaded immediately, using the fastest possible methods.

The basic functions of the cache are:

- **Lookup Resource** – You give this routine a resource id and it attempts to find it in the cache. If it's there, you get back a pointer to where the resource can be found in the cache. If it's not there, someone will have to request that it be loaded and then try again later. It's also possible that this routine could be setup to not return till the resource is loaded, this could cause serious frame rate stutters, but if the predictor code is doing it's job, this kind of "cache miss" should never happen.
- **Load Resource** – A request is queued to transfer the resource into the cache. In the case of a texture RAM cache this would queue up a request to move a texture from system RAM to texture RAM.
- **Purge Resource** – Removes a resource from the queue, used when your game knows a particular resource won't be needed anymore.
- **Clear Cache** – Used to clear the cache completely, or to clear out all items in the cache that haven't been used in a certain amount of time. Used occasionally, when the game has some spare time, this can help prevent fragmentation of the cache.

Prediction

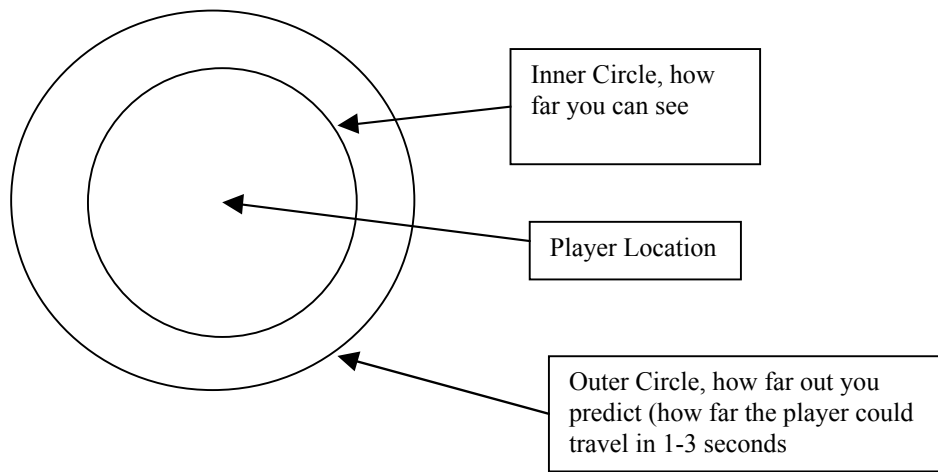
For this kind of resource caching scheme to work well, some kind of prediction system is required. Without the ability to predict and start things loading in the background well in advance, you'll still get frame rate glitches whenever a new group of resources are needed, because they will have to be loaded immediately instead of over time. Many people have avoided prediction because they think it is too complicated to do in a real-time game. Actually, with proper preparation of your data the real-time prediction can be very simple and fast. A very large amount of the work of prediction can be done off-line when the game is being developed and mastered onto the CD. In fact, the biggest challenge in doing prediction is finding efficient ways to store the data so it won't take up too much space. Even relatively brain-dead prediction schemes can greatly reduce frame rate glitches and the better your predictor works, the closer you can get to the edge of available system resources and still function well. A good predictor will let you run well on more limited systems and will allow your designers to be much less concerned about how the game engine works when designing worlds.

One thing you should keep in mind is that prediction is very sensitive to the speed that the player can move through the world. If the player has the ability to move at very high speed, turning very quickly, it will be much more difficult to predict far enough ahead to avoid frame rate glitches.

Simple Real-Time Range Based Predictor

This is a simple type of predictor that does all it's work in real-time, it's suitable for some kinds of games where the world isn't too complex, particularly ones set in outdoor worlds. In this case, you scan the 3d world for every object enters a specified range (normally as far as the player can see, plus what they can move in a few seconds) and issue load requests for any resources required to render those objects. You can also track objects as they pass out of range and unload their resources to make room for new ones. The problem with this approach is that it does all it's work in real-time, so the

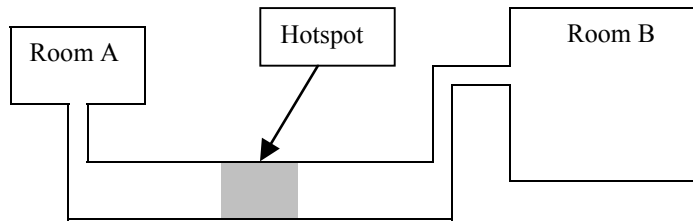
more objects you have to scan, the more time it takes. Because of this it's only suitable for fairly simple worlds. It also doesn't have any way of taking advantage of things that block your view (like walls) so it's not suitable for indoor worlds.



Hotspot Prediction

In hotspot prediction, you place trigger locations in the map (either manually, or with an automatic tool) when the game is being designed. Each hotspot has what amounts to a list of resources that should be loaded when the player enters it. This approach works pretty well, but the list of resources for each hotspot can get kind of long. Also, figuring out what resources should be loaded for each hotspot is tricky. For example, if you're in a hallway leading to a cathedral, a hotspot in the hallway might trigger the loading of the cathedral's resources so they will be loaded before you get there. Unfortunately, since you're not taking the player's direction of motion into account, you may also trigger loading of these resources as the player leaves the cathedral through the same hall. Again, this isn't the best approach for an indoor game. It can work very well for an outdoor game though, particularly when you have to swap between various detail levels of models. A hotspot can be placed around a building, for example, that triggers loading of a more highly detailed version of that building's models and textures. This makes sure the more detailed version of the building will be loaded and ready for display before the player gets too close.

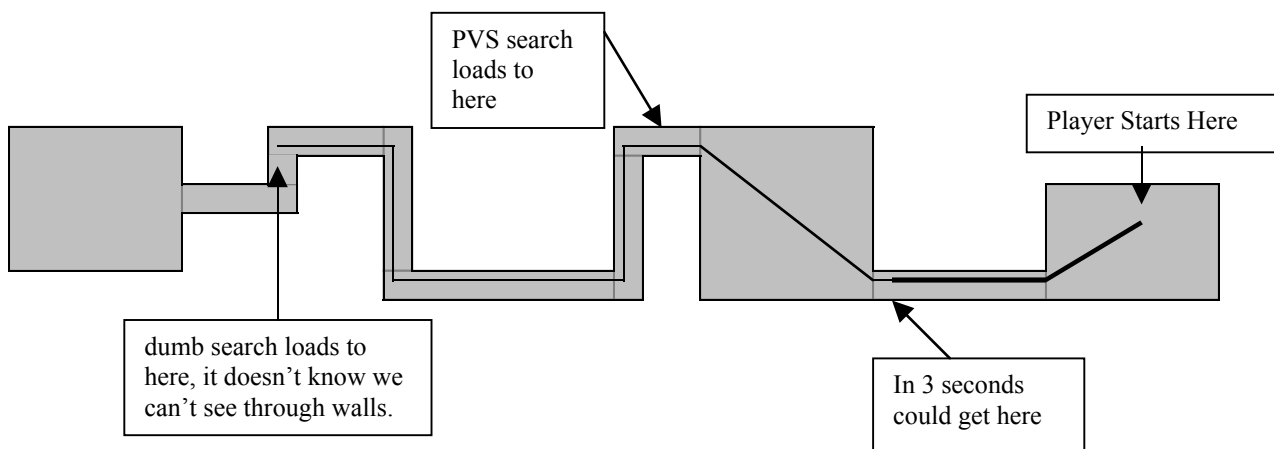
You can improve this technique and make it work better for indoor games by having a hotspot that takes the player's motion into account. In the figure below, a player passing through the hotspot moving to the left would trigger loading of the resources needed to draw room A. One moving to the right would trigger the resources for room B. This can work very well for indoor games, particularly if they contain a lot of hallways where hotspots can be easily placed.



Search Based Prediction

This is the most flexible type of prediction, it involves doing a fair amount of processing off-line and a fairly minimal amount when the game is running. It works very well for indoor games where hallways, doors & such really limit what you can see. For this approach you grid out your 3D world into zones. A room might be one zone, a long hallway might be broken into several. You also need to know how these areas are connected (doors & windows) so you can figure out what zones can be seen from other ones. With this method, each zone contains a list of all the resources needed to draw that zone, and links to nearby zones that you could see from inside that zone. If the zone has more than one way in and out of it, you will want to store a “shortest path” through the zone to use when doing your predictions.

To run a prediction, you first figure a “range” that represents how far the player can see plus the distance they could potentially cover in 1-3 seconds (or longer if you have a lot of RAM free). Then you do a search starting from the zone the player is in and moving outwards. Each time you pass through a zone you subtract the shortest path from the prediction range. You stop searching when a path reaches a dead end, or the “range” runs out. When the search is done you load the resources for every zone you passed through. This guarantees the resources for every area you could get to in the next few seconds and every place you could see from those areas will be loaded. The only problem with this approach is that doesn’t take very good advantage of things that might restrict the players view, like curving hallways, so you end up searching a wider area of the database than necessary. In the example below the heavy line indicates the distance the player could cover in the next few seconds, and the light line recognizes the distance the player could see. If the player had a long hallway to look down, they could see pretty far, but in this case the hallway is twisty, so we end up looking ahead much farther than we need to.



Potentially Visible Set Based Prediction

Many game engines such as Quake use a technique called the “potentially visible set” (PVS) to minimize drawing invisible polygons. This technique chops the world up into zones (usually with a BSP tree or something similar) and calculates for each zone what other parts of the world are visible from inside that zone. This visibility calculation takes into account things like walls that obstruct your view of other parts of the world. You could calculate what zones are visible from each zone, or what polygons are visible from each zone. The first approach is easier to do but the second works much better if your world has a lot of complex geometry. This data is then used to make sure only polygons visible from where the player is standing are drawn.

A few small additions to the potentially visible set technique let you use the same data and calculations for prediction. As you calculate the potentially visible set data, also create a list of the resources that need to be loaded to draw what’s visible from each zone. This greatly reduces the number of zones you have to search through to do a prediction. You only have to search zones the player could get to in the next few seconds, not beyond.

This is probably the most efficient technique for prediction of resource needs, and has the added benefit of producing data that speeds up your renderer as well. Because it’s a very accurate prediction technique it allows you much more freedom in designing your 3d world. In indoor spaces the subdivision of space can usually be done with BSP trees based on the walls of the rooms. Outdoor spaces are a bit more of a problem, unless your world is designed to give the player a limited field of view. An area like a forest would be difficult to be effective in, but a downtown city would work fairly well, particularly if many of the streets were curved or there were hills to limit the field of view. A good example of this kind of arrangement would be San Francisco, Boston or London. If your game allows the player to get high enough to see over all the obstacles, this technique could break down severely as they can see the entire world all at once.

In the previous figure, a PVS based prediction scheme would take into account the fact that the player can’t see around corners and would only load the resources up to the spot I’ve marked. This may not seem like too much difference, but if there were many branching hallways or rooms coming off the hall, it could save you a lot of unnecessary data loading.

The figure below shows a more detailed example of how a PVS can reduce the amount of data you have to load. From where the player is standing, the shaded areas would be the PVS. If you were using a simple search technique that didn’t take visibility into account, you would have to load almost everything in this diagram

Preparation

If you want to deal with a very large 3d world in a seamless fashion, the most important thing is to properly prepare all the resources (textures, geometry, sounds, scripts, etc.)

in advance to enable them to be found and loaded as quickly as possible. Most data will already be in reasonable size pieces for loading but will need to be indexed. Some data, such as the geometry for a major section of the game, may need to be broken up into smaller pieces so it can be loaded a bit at a time as you need it.

Make sure all your data is formatted in ways that allow fast loading. Don't store scripts, geometry and other data in text formats that need to be parsed when loading. Convert as much data as possible into formats you can simply read directly into in-memory structures with little or no conversion.

Data Dependancies

The next thing you want to do is calculate dependencies so that, at runtime, you can do fast prediction of what data needs to be loaded at any particular time. What works best for you will depend largely on what type of game you're doing. For example, in a Quake style game you would want to know which textures are associated with each room, what objects are in the room (for example what furniture goes in what rooms) and other information. You can use a lot of different techniques for organizing this information, but trees are probably the easiest. Once you have this cross-index of dependencies, you can use it to organize your data so that related resources are stored close to each other on the disk for quicker loading. You could put all your resources in one big file, but if you have a large amount of information you might find it improves performance some if you group data into several separate files.

If you organize your dependency information properly, it can streamline your prediction and resource management system considerably. If a room has links to all the resources needed to draw the objects inside it, then all the predictor has to do is tell the resource management system to load dependant resources for that room. This greatly reduces the amount of data the predictor has to deal with and prevents it from making a large number of load requests to the resource manager.

Texture pre-conversion

One other obvious bit of preparation you can do to help runtime performance is to pre-convert textures. When your program starts, get a list of all the formats the current 3D card supports and compare it to a similar list from the previous startup. If the two lists don't match (or it's the first time the program has run), it means the 3D card in the system has probably changed. If any of the textures in your game need to be converted to other formats for this card, do it now and save the converted versions. This will allow textures to be loaded much faster during the game since no conversion will be required. Remember that all texture formats are not necessarily equal in speed on all 3D hardware. Some hardware may have a speed penalty for using certain texture formats or speed bonuses for using compressed textures. Unfortunately, most 3D APIs give you no way to determine this. The only thing I can suggest here is to determine which formats are best for which cards during your testing process (or ask the manufacturers) and, where necessary, store a list of the best texture formats to use for cards that need

it. If your 3D card and API supports directly loading compressed textures, compress them in advance and store them that way. Since compressed textures are smaller, they will load from disk more quickly. This is also a good time to look at the amount of available texture RAM on the 3D card and decide if you are going to need to downsize some of your larger textures, convert rectangular ones to square or do any other kinds of conversions. The whole conversion process will probably only take a minute or two and shouldn't be objectionable to the user since it won't happen again unless they change 3D cards.

Resource Numbering

When working with large databases it's very important to be able to find the data you need very quickly. You need to be able to locate a data item in any of several indices/caches very fast, because you are going to have to do this operation a lot. I recommend that you assign all resources consecutive ID numbers and have carefully optimized routines finding the ID numbers in the various lists you keep. The number of resources you have and what percentage of them you need immediate access to will determine the best way to index them. Direct indices (where you use the ID number as an array index to find that item's data) are the fastest but may become too large in some situations. Also consider sorted arrays that can be binary-searched for ID numbers. If you do end up using a search technique, you might want to use a type and subtype system (ie: type 1 is textures, type 2 is geometry...etc) this breaks up the indices into pieces that are smaller and quicker to search.

Efficiency Considerations

Virtual Memory

When possible, you want to try and arrange your data so the operating system can manage it more efficiently. Find out things like the size of virtual memory (VM) pages and how memory allocation is done, then arrange your data on disk and in RAM to take advantage of this. For example, windows virtual memory currently uses a page size of 4K. If you try to align your structures in memory so they don't cross these boundaries, windows will be able to manage them more efficiently. For example, if a structure that crosses a 4K boundary in memory needs to be paged in or out by the operating system, a total of 8K of data will need to be moved. If it doesn't cross the boundary only 4K will be moved.

Many game authors try to avoid virtual memory paging because they think it will slow their game down. Because of the hardware support most CPUs have for VM, this isn't strictly true. If you control your use of VM properly it can be the fastest way to manage paging data from disk to RAM. The problem is managing it, since few operating systems for PCs or consoles offer good, fast ways to either control the VM system or see what it's doing. The key to good use of VM is organize your data in RAM so information that is used together, is kept together. If you keep related data together the

VM system will have to load a smaller number of pages. If you don't, there will be a lot of 4K pages loaded where you only use a small piece out of each one, reducing the amount of memory left over for other things.

CPU Caches

Just about all modern CPUs have some form of memory cache between the CPU and main memory. While you can't control this cache directly, you can setup your data to take best advantage of them. Typically this means aligning data structures to cache boundaries, keeping data structures small so a lot of them will fit in cache at once, keeping related data close together in RAM and making sure data structures are packed together with a minimum number of gaps.

Because the techniques in this paper involve a lot of searching of various data structures, you will want to make the search code as cache friendly as possible. Aside from just keeping structures small and close together, you may also want to separate the index to a structure from the structure itself. For example, if you are searching by resource number, you might want to have one "index" array consisting of only a resource number and a pointer to a structure where the rest of the data is located. If you allocate this index array as a single block of RAM, the small size of each item in the array will insure that a lot of them will fit into the CPU cache, making searching or frequent accessing of the array much faster.

Intel's Vtune utility is the best way to find out how cache efficient your code is, it can profile areas of running code and show you how many cache misses occur in it.

Memory Allocators

As I've mentioned before, it is always best to keep related information together in memory. If you are using C++ one way you can do this is by having a custom allocators for each C++ class. This allocator can do things like grab memory in large blocks, several times the size of the actual class, then putting future instantiations of the same class into the same block, keeping them together. This isn't always the best way to do things, but if you don't have the time to come up with a more complex scheme to keep related data of different types together, this approach works quite well. It's particularly good for allocating frequently searched data such as indices, resource caches, trees or linked lists. If you're not using C++ you can still do this, it's just takes a bit more work.

Be particularly careful of how you allocate RAM for linked lists, trees and similar linked structures. If you use malloc to allocate these structures as needed and free to remove them, you can quickly end up with structures scattered all over RAM. This will lead to poor performance because of poor use of the CPU cache and virtual memory. If you don't do it anyplace else, it's good to use custom memory allocators that will allocate these structures from pools of contiguous RAM.

Another technique you might consider is trying to load ALL the data associated with a particular object (like a room or vehicle) into one big block of RAM, insuring it stays together. To do this you'll need to be able to figure out how much RAM all this data needs, allocate a big block and then load all the different data types into this block. This is a lot trickier to do and can give you problems if you want to purge just parts of this data.

Files

It's best to keep the number of files your application opens relatively low and keep all of them open at the same time. Opening a file typically has a high overhead and isn't the sort of thing you want to do when you're trying to render at 30 or 60 hz. Keeping the number of open files low also allows the operating system's disk caching system to operate more efficiently. If your game has several natural breaks where the player is pulled out of the 3D world, you may want to have one set of resource files for each section.

Some games require the ability to write some of their resources back to disk, this might be used to save back textures that you've painted bullet holes on, or geometry that you've "damaged" and want to appear the same when the player sees it the next time. If you have writable resources it's best to keep them in a separate file from the read/write ones. This simplifies some of the caching code and can allow you to use memory more efficiently, particularly if you use memory mapped files to manage some of your resources.

Profiling & Time Management

Taking measurements

You've probably noticed that almost all of these techniques involve the resource management system doing a lot of work in the game's "spare time", when the game has finished what it needs to do for the current frame and is waiting for a page flip. Obviously, for these techniques to work, you have to keep close track of what your game is doing all the time so you know how much free time you have to work with. It's also important to track the time used by the resource management system because it's easy to accidentally make a system that's too complex for it's own good, and spends more runtime than it saves.

To do this you need to take frequent time measurements of various parts of your game when the game is running. The best way to do this on a Pentium II class machine is to use the read time stamp instruction which fetches a CPU clock cycle counter directly from the processor. This gets a timer that has the same resolution as the CPU clock (ie: if you're on a PII 450 it counts at 450mhz). If you're on an older computer, you can use the "get performance counter" routine from the standard windows libraries. This call takes a bit longer and doesn't return as high a resolution time (roughly 1mhz). Intel

offers a library with a routine that fetches a time value for you however it can, depending on what CPU you have (it also returns a lot of data on CPU type and ID). The routines in this library are very general purpose but a bit slower than hard coding it yourself, however they do return usable timers regardless of what CPU you have.

Whatever technique you use for measuring time, you are bound to have some timers intended just for tuning the code that you don't want in a production build. I would suggest wrapping the timer calls with macros so you can easily compile timers out of the code.

As the game starts

When the game first starts up, you want to collect some information how it performs so your resource management system will know what it has to work with. This information doesn't have to be too extensive, but I recommend you get some idea of available RAM, texture RAM, texture load speed and how fast you can load data from disk or CD. It's also important to try and find out what refresh rate (monitor frequency) the video card will be running at during the game. This lets you synchronize up your time management system with the monitor.

If you use in-engine animated sequences in your game's startup or "attract" mode screens you may want to run these tests while the 3D engine is running. Measurements taken this way will be a better reflection of the performance you will get during the game.

While the game is running

At minimum you will want to keep track of your frame rate, you may want to separately track such things as 3D, AI, networking, resource management, prediction and any other part of your program that burns significant time. This may seem like a lot, but it's probably less than a dozen timers. If you are on a Pentium II or better, these time measurements can be taken so quickly it won't slow the game down at all.

For best visual quality and best operation of the resource management system, I would recommend the game always be run with "page flip on vblank" turned on and that you construct the game to run at as stable a frame rate as possible. This doesn't mean your frame rate can NEVER vary, you just want to continuously monitor your timers to pick a stable and achievable frame rate that may vary slowly over time. By sticking to this stable frame rate you can guarantee that at least some frames will have free time for your resource management and prediction system to work in.

Games are running at increasingly higher frame rates and at these speeds it is much more important to keep your frame rate steady and keep it in sync with the refresh rate of the screen. At high frame rates, the transition between two adjacent frame rates can be huge and very jarring, like the 2X difference between 30 and 60 hz. So by using these timers to stabilize your frame rate, you make your game look better as well.

Cheap Tricks

Some areas of your world will undoubtedly stress the prediction and loading mechanism and you'll find that it just can't get all the resources loaded in time. Sometimes the player will be moving through the world so rapidly that the prediction and loading mechanism simply can't keep up. Other times the part of the world the player is in may be so complex to render that it doesn't allow the game enough free time to perform the loads. You can also have these problems if the player is moving between two very complex areas, each of which barely fits in the available resources.

Obviously, one way to get around these problems is to simplify key areas of the maps so you don't encounter these extreme situations. Unfortunately, this can make your game kind of boring and may force you to lose some of the more dramatic moments like long hallways that lead into a grand space or sudden shifts in architectural style. Luckily, there are a variety of tricks you can use to make the prediction process easier and to control the player so you can keep these dramatic moments in the game and still avoid frame rate stutters.

Idle zones

An idle zone is just a section of the map that can be rendered easily, giving the game engine more free time between frames to deal with the job of predicting, loading and purging resources. The classical technique is an air-lock where the player has to pass through two doors, temporarily locking him into a very simple stretch of hallway between the doors. If your prediction and rendering system is fairly smart, you don't need to be quite this obvious. An idle zone could be something as easy as a stretch of simple corridor or a few simple rooms the player needs to pass through on their way to a more complex space. If you have a good potentially visible set style prediction system, a stretch of hallway without doors will work just as good as one with doors at each end.

Elevators

Elevators extend the concept of idle zones further. Because players expect to be locked in an elevator for a bit of time, you can easily use them to insert fairly long pauses during which the caching system can catch up. Since the inside of an elevator is very simple thing to render and the player won't be moving much while inside of it, there will be quite a bit of free time for your engine to load data into the caches. Also, since you know exactly where in your world the elevator can stop, you can have extensive pregenerated data describing what data needs to be loaded and purged from the caches while the elevator is in motion. For example, you could force out all the data associated with the first floor of a building and replace it with data for the second.

Doors

Doors are another way to slow the player down a bit and give the engine a chance to catch up. Even the relatively short delay caused by waiting for a door to open can give your engine the time it needs and since the player will usually be looking straight at the door you will get at least some fast rendering frames with free time for resource loading. You don't want to overuse this trick though, as having the player pass through too many doors will seem unnatural. One way to reduce frustration is to use doors the player can see through, and possibly shoot through. This provides the pauses you need without giving the player the impression you're deliberately trying to limit their view.

Announcements

Various types of voice announcements can be used to get the player to pause in a particular area. The easiest way to do this is have the announcement come from a particular location (like a wall speaker) and be quiet enough so the player can't get too far away and still hear it. This will tend to hold the player in that spot till the announcement is over, giving your caching engine a chance to catch up. Psychologically speaking, when most people hear an announcement, they instinctively stop to listen, so playing even a non-spatialized voice announcement could slow a player down a bit.

Virtual Memory

If your game is running under an operating system with virtual memory, you may be able to use it as a combination RAM and disk cache. To do this you simply allocate as much RAM as you need to hold all the data you expect to use in the course of the game. When a piece of code tries to use the data, it will automatically be loaded from disk to RAM if needed. The advantage of this approach is that it's very efficient, using low level hardware to manage keeping the most frequently used blocks of data in RAM at all times. The disadvantage to this approach is that there's no easy way of controlling when data is paged in or out of RAM. This can lead to nasty pauses at inappropriate times. You can reduce this by having your predictor "touch" RAM blocks in advance of when they are actually needed, forcing them to be loaded into RAM, but in the end, you're at the operating system's mercy. Also remember that any memory you allocate will eventually be copied into the system swap file, so if you allocate a large amount of memory you could end up filling up the hard drive with a very big swap file.

Still, if your application is not overly demanding, this approach can work quite well and automatically makes use of all the available system RAM. I've seen this approach used where the main concern of the developers was managing large amounts of textures and it worked quite well.

Memory Mapped Files

If your platform supports memory mapped files, these can also be used as a shortcut similar to Virtual Memory. Memory mapped files use the virtual memory system also, but have the advantage that they can be setup so writes to them will be stored back to

disk. This is very handy for data you want to preserve from one run of the game to the next. Another advantage is that the way you organize the data in the memory mapped file when you create it is the way that data will be organized when it's mapped into system RAM. This allows you to organize the data structures in advance to take best advantage of the CPU caches.

When using memory mapped files you will want to separate read only and read-write data into separate files since a read only memory mapped file uses a more efficient form of memory management than a read-write one.

Levels of Detail

A really good Potentially Visible Set prediction technique can work so well that level of detail handling is almost unnecessary. However, if your game lets the player get into a position where they can see a lot of the world at once (ie: up on a hilltop looking down on San Francisco) you can end up with more data than can be loaded. If your game allows this to happen, then you need to use "level of detail" techniques. Traditionally, this means having several successively simpler versions of every object in the game and using the simple ones for objects that are farther away. Unfortunately, in the case of a complex model like a city, this may not be enough. To get things to fit into memory and draw fast enough you may need to take a bunch of objects that would normally be separate models and combine them into one. If you are using the PVS technique, this is easy to do. When down at ground level in a city, you do things the way you would expect, and the various buildings blocking your view keep the amount of data you have to deal with reasonable. When the player goes up to the observation deck of a tall building, we cheat. Instead of having the PVS for the observation deck include the whole city, we link it to a simplified model that's designed for rendering from this point of view and this distance. Because we know the player has a limited point of view, we can simplify the buildings greatly, dropping out much of the detail (and the backsides) of the nearby buildings and replacing the more distant ones with flat textures of the skyline.

Resource Numbering

I've suggested you give all your resources consecutive numbers in several places to make the process of searching for a resource faster. Some people have probably tried this before and found it can cause problems during the development process. Even if you have tools to manage this job, it's possible that resource numbers could get out of sync with data in other files or in the code itself. There are many ways to get around this problem, you will have to give your development process some thought and come up with one that works for you. One of the easiest things to do is to setup resources with fixed, hand-assigned numbers that are not consecutive. This assures the resource numbers will remain the same throughout the development process but requires you to have a scheme in your resource manager code that can deal with large gaps in resource ID numbers. When the development process is mostly complete, you can switch to consecutive resource ID's and renumber everything with a tool.

Conclusions

By using these techniques you can greatly increase the size of the 3d worlds of your game and allow the players to move freely through them without distracting “loading please wait” pauses. A good resource management and prediction system can allow the use of a great many more textures and geometry so you can have large shifts in architectural style as the player moves through the game. With this kind of system, it’s less necessary to re-use textures and geometry to save RAM, so players won’t keep seeing the same identical vase, chair, lamp or other objects being repeated in room after room of the game.

In addition to these improvements, resource management allows your game to perform well on a wide variety of computer configurations. It will run well on a low end machine while still taking full advantage of all the RAM, disk and other facilities of a high-end gaming enthusiast system.