

Intended Audience:

Primarily, Game Engineers who write game tools and runtimes, and Technical Artists who work on pipelines

Originally Published on *Gamasutra.com*

May 1, 2008: [Part 1](#)

May 15, 2008: [Part 2](#)

## *Building a Mindset for Rapid Iteration*

*By David Gregory*

### **Part 1: The Problem**

#### **Why Focus on Rapid Iteration?**

You often hear from people who build games for a living that it's unlike building any other piece of software. Why is this? Building interactive fun is a very different problem than building any other type of software, because "fun" can't be planned: you can't schedule a project with waterfall, execute the schedule perfectly, and expect to get a fun game 100% of the time at the end. There are usually lots of go-backs and retries as you bring together elements of the game – the code, content, scripts, etc. – and see if it's entertaining. In the worst cases, teams don't realize that it's not a fun game until it's so late, they can't fix it. In the best cases, teams prototype their game in 'sketch' form during pre-production, and find the fun before they spend the bulk of their time in production filling in content and polishing the game play.

So, how do you align your team with the best case scenario? It's a generally accepted practice in the industry that quickly iterating and trying different things is the best way to make incremental progress towards the final goal of a fun game. It's less risky, and you find the core of your game (look, mechanic, etc.) much more quickly.

#### **So What's the Problem? Why Can't I Rapidly Iterate?**

Developers are now dealing with *more* - more of everything. For instance, these elements have all increased:

- People on a game team
- Specialization of talent because of complexity of technology or tools
- People necessary to iterate on a single game element
- Number of team members distributed across multiple geographies
- Amount of time it takes to compile and link code due to increased size of code
- Amount of custom tool code
- Amount of content needed to satisfy today's players
- Complexity of content creation and transformation
- Number of tools to create and transform content

- Amount of time it takes to transform content
- Number of platforms simultaneously released from the same team (PC, Xbox 360, PS3, Wii, etc.)
- Infrastructure (code & content repositories, automated build farms, automated test farms, metrics & analysis web sites, offsite development infrastructure [VPN, proxy servers], game and tool build distribution, etc.)
- Rate of change of content, code, tools, pipeline, infrastructure, etc.
- Build stability problems because of all of the factors stated above
- Dollars at stake when a mistake happens that hurts productivity due to number of people idle and unable to work
- Management focus on risk because of all the factors stated above

That last one (increased Management focus on risk) has created a cyclic dependency in some of these items that have actually increased them even further, particularly infrastructure. Increased focus on risk brings with it the wish to control the chaos, and implement systems that provide increased visibility and predictability into that chaos. Game team management usually doesn't have enough information to know where the problems are. Getting the information in place requires new systems, and hooks into existing systems, which increases the rate of change of code, the amount of code, and the number of systems to be maintained. Game build systems will usually be put in place to increase the predictability of new releases back to the team. When those aren't stable, other things are done to make those systems more reliable. Everything done here adds to the volume of work and the complexity of the project.

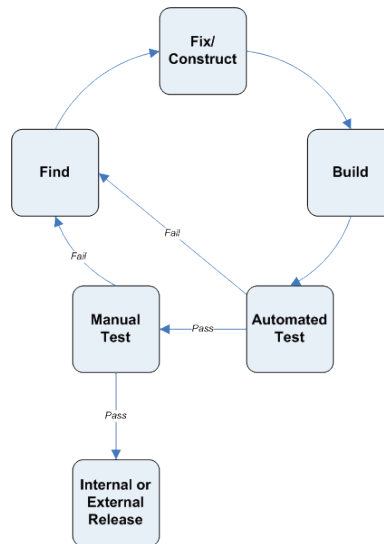
As game development has gotten more complex, projects have become unwieldy and for many teams, progress has slowed. It's caused developers to have a focus they never had to have before: optimize not only the game, but also the tools and processes involved in building the game. It's a huge shift in mindset.

## **Local/Individual Iteration**

What items above are most responsible for impeding a single developer's progress these days? It's generally the tools involved in daily workflow: compilers, debuggers, digital content creation tools, source code and content control systems, custom tools and the game runtime itself. Each component in the tool chain adds time to your iteration rate. When you make a change, how long does it take you to see the effect of that change in the most appropriate environment (game, viewer, whatever)? The closer it is to instantaneous, the better you are doing.

## **Team Iteration**

A change made locally doesn't really exist until everyone on the team can see it (*Is it in the build yet?*). This is usually all about getting your change submitted to a repository, and getting it successfully through a build cycle. Here's a build cycle diagram that will be useful in discussion.



As project leaders and studio executives have trouble understanding how far along their game development project is, they implement metrics and automated processes to help them achieve some predictability. They stand up automated builds to bring order to a very complex, error-prone process.

In addition, to stabilize the build and protect the ongoing investment in the game, “gates” are placed in the way of checking in anything into the game. Sample “gates” include:

- Get your change integrated into your local build
  - If it’s a code change, either:
    - a. Compile a single configuration on your main development platform.
    - b. Compile all configurations (debug, release, shipping) on your main development platform.
    - c. Compile all configurations on all platforms.
  - If it’s a content change , either:
    - a. Transform data on your main development platform.
    - b. Transform data on all platforms.
- Test your change by either:
  - a. Running manual tests on all platforms, performing sufficient tests to verify that your change worked as expected and didn’t break anything.
  - b. Running an automated test on all platforms, in addition to the above.
  - c. Running the entire automated test suite on all platforms, in addition to the above.

*Wow. I’m forced to do that to check in my change? No wonder things are slow.*

## What Do I Do to Fix This?

The best approach is to treat this situation like an engineering optimization problem. You have to measure the process to find the bottlenecks, pick some low-hanging fruit to go after, and then tackle the tougher problems. You should start with each individual's process, and then look at the team as a whole. Never do something with one without considering the impact on the other.

One place to start is with the artists' tools. These days, there are more of both artists and tools, so fixes there will have a big impact.

Some content transformations are more expensive than others. "Expense" is defined by the *time elapsed before the change can be seen in the appropriate medium, usually a game engine or an engine-derived viewer*. Factors contributing to this elapsed time are transformation, hard drive read/write bandwidth, processing power, memory size, memory bandwidth, and network bandwidth. The key to *rapid iteration* is reducing this expense.

People say this, then forget it, so it's worth mentioning again: Measure first, then optimize. Don't guess at where your bottleneck is.

As part of looking at your iteration bottlenecks, you should try to classify the different types of data transformations you have in your pipeline. This can help you understand what's taking the time, and what changes are possible – and safe – for you to make.

## Types of Content Transformations

Description	Example
Data reformatting	Generic mesh to tri strip
Mathematical	3D math, vector math, animation curves, shadow maps, baked lighting in textures, image compositing
Bit Twiddling	Image format changes (bit depth, alpha, mip maps, HDR)
Copying/Simple Packing	No expense other than copying the data from one file to another, or from one place in memory to another.
Duplication	Putting the same object in multiple streams or sectors of the world to speed up runtime loading.
Compression	DXT texture compression. There are cheaper compression algorithms, but you usually trade increased compression time in the pipeline for reduced load/decompression time in the runtime. DXT texture decompression is hardware assisted, so it's very cheap in the runtime.
Decompression	Lots of data coming from art packages are stored in a compressed form. TIFF texture files, in some cases, are compressed.
Level of Detail	Stripping/adding data. Server doesn't need graphical data, but does need collision data. Clients may need several levels of detail.
Compiling	Converting from human readable text to some "quicker to load and execute" form.
Complex Packing	Some intensive process is required to layout the data into a file or set of files. Setting up files to optimize seek times based on world layout (streaming) can be very expensive. Especially with respect to the size of the actual change.

## Managing Expectations

For some content transformations, the size of the change is reasonably proportional to the "Expense".

Developers are usually most upset when they perceive they've made a small change to source content, and see a disproportionate expense. They'd be even more upset if they additionally found out that only a few bits of game-ready data changed after all that time.

By the same token, even when the size of the change is small, some types of transformations take an incredible amount of time and require nearly the entire game world to do their job, such as stream packing steps, global illumination or light baking builds. These need planning so they have a minimal impact.

If you take a methodical and holistic approach to optimizing your pipeline for rapid iteration, you'll see great results at the individual and team levels.

**Coming next:**

## **Part 2: Some Patterns to Follow and Pitfalls to Avoid**

### *Building a Mindset for Rapid Iteration*

*By David Gregory*

#### **Part 2: Some Patterns to Follow and Pitfalls to Avoid**

In Part 1 of this series, we discussed the reasons why rapid iteration is so critical to your chances of success in building fun into your game, and some of the contributors to increasing iteration rates as teams, projects and toolsets grow ever larger. Content transformation "expense" was defined as *time elapsed before the change can be seen in the appropriate medium, usually a game engine or an engine-derived viewer*. With the goal of increasing efficiency on a game project, we started by looking at content transformations as the first optimization point. Now let's dive into the details of where you can squeeze significant time out of your processes: the development team's tools and practices.

### **Patterns to Follow**

If the whole is equal to the sum of the parts, then the iteration rate for each individual developer on the team makes a big difference in your overall iteration rate. Make sure that each developer is working in the most optimal environment possible.

## **Get a Handle on Your Development Workspace(s)**

Maximizing productivity is a lot about the details of a developer's day. Minimizing disruptions is important, be they attendance at unnecessary meetings, or just interruptions that break the flow of concentration in the middle of a task. For example, you need to be able to Context Switch between development workspaces quickly, on the same machine. You may be asked to work on a feature, and at a moment's notice, fix a bug you aren't set up for. How do you minimize the interruption?

The development workspace is the collection of data, software, tools and utilities that achieve a number of transformations on data. For instance, your compile workspace includes source code, compiler, linker, environment variables, registry entries, project files, solution files, etc. Your artist workspace includes digital content creation tools, the last known good pipeline tools for your game team, and the last known good target environment for you to check out your work in.

A lot of workspaces are set up as global singletons, making it impossible to switch workspaces on a single machine. This makes it very hard to set up your machine to debug a problem from another branch, and it makes it very hard to keep your build process the same on both your local machine and the build farm. Notorious "problem child" software includes anything requiring an installation procedure, or anything setting up registry entries or global environment variables. This includes anything installing itself in the global assembly cache. Any configuration with hardcoded drive letters, or absolute paths, is generally a no-no.

The best configurations are usually file-based and script-based, and can easily be moved around from base directory to base directory, and can be distributed simply by syncing from a source or content control repository, or from another distribution mechanism if required.

## **Plan for Team-Wide Change**

Game development is all about dealing with change. Development teams will often customize anything they can to achieve the desired result. There are some things you can do to try to limit the risk of making those changes.

For one, you should reduce your variables when introducing new functionality in the pipeline. Change as few things at a time as possible. That means staging your changes and lining up test plans for each change. For example, you should not introduce a new texture compression method to your texture packing tool at the same time you're adding cube maps. Let one get tested and introduced to the team before the other is rolled out.

You should also set up a way of reliably testing new pipeline tools without affecting the entire team. Your content build farm should be able to run with last known

good tools, or with untested tools. Results of an untested tool content build should not be released to anyone other than the person(s) testing the results.

Remember that not everyone is using the same versions of data and code. You should introduce data versioning and serialization capability into your non-shipping application. Train your engineering staff to maintain version compatibility at least one version back from the last known good build, or whatever range you are comfortable with. This can be achieved without affecting the performance of the shipping application.

## **Follow Good Configuration Management Practices**

Your pipeline should have a good Audit Trail, with logs, metrics and content lineage. You must be able to easily tell the difference between the data you build locally and the data you get from a build system. Don't commingle. Compartmentalize your data.

## **Reduce Compiling and Linking Times Where Possible**

Compiling and linking times are huge on projects the size of a modern game development effort. Do what you can to shorten these times.

There are lots of tools available to engineers to make things go quicker; Pre-compiled headers, incremental linking, and distributed build systems, to name a few. *(Note the pitfalls below.)*

## **Perform the Smallest Number of Transformations possible**

Understand the roles on your team, and what build results each person MUST see to determine if their change worked. For example, do you need animators to run your lighting build?

## **Transform Only What You Change**

Transform the minimum amount of data that you can get away with. If you edit a texture, transform the texture, and nothing else. In fact, if you only need to change the alpha channel of the texture, make sure your pipeline supports outputting only the alpha channel (assuming you store it separately).

Do the minimum amount of data transformation (measured by time elapsed) that will still put the data in a form that the target application can read. Decide what's important that you see on the target. Is real lighting important for every change made? If you eliminate some steps, can you still do your work?

Don't do dependency-based content builds during host-target iteration. It will not scale. Large amounts of content will take forever to check. It's better to intelligently build only what has changed. Your "do nothing" build should literally be, do nothing. Note that you likely *want* a dependency-based build in a build farm, where

you are building lots of content, have all of the intermediate data (hopefully local to the machine that needs it), and when an iteration time measured in seconds is not required.

## **Location Independence and Format Flexibility**

Your game engine (in non-shipping form) should be able to read data in a shipping/optimized form and many non-shipping/non-optimized forms. It also should be able to read it from different physical locations.

For example, optimized stream creation can be very expensive and require tons of source content and/or intermediate data to be available. It will always be better to be able to read the data without ever doing the stream creation. Be able to retrieve records from several locations on the target and from remote locations, and have this be transparent to the upper layers of game code.

## **Load Only What You Need**

Game engine load times are bad, especially during development. Viewing tools sometimes have bad load times too. Pay for the load once, and then load only the data that has changed.

In addition, to make it easier to “try things out”, get a network connection going between the host and target, and set up a reflection mechanism so you can change data in running objects on the target without changing what’s on the hard drive. This is great for particle system tuning, sound tuning, and in lots of other situations.

## **Keep the Data Moving**

Always enforce a preference of memory over disk over network, but above all, *measure!*

For example, don’t write data out to disk with one custom tool, only to read it back in with another custom tool – when the two tools might have been architected to work together and act on the data entirely in memory. Sometimes this is not possible because an external tool is not architected to allow the data to move in this manner. However, when you’re building your own tools and are in control of your own destiny, make sure you don’t “leave money on the table,” so to speak, by overlooking these important opportunities to optimize.

## **Only Compress and Compile When You Must**

Compression and compiling can be time-intensive functions. Assume that your target application can handle an uncompressed or uncompiled version. If loading the raw form of data into the target application is shorter than the compression or compiling time, plus the time to load the compressed or compiled data, then you’re better off not compressing or compiling in the first place. This goes hand in hand with format flexibility, as mentioned above.

## Reduce Complex Dependencies

The more data you require to perform a transform on your machine, the more likely it is that you will have to retrieve or build that data to perform the transform.

This is sometimes unavoidable, but you should try to minimize it wherever possible. This comes into play particularly in data packing steps, where many records are packed into one big file. Avoid data packing steps during host-target iteration.

There's another way to look at the same thing: from a hard drive perspective:

Size of data written = Size of data actually related to change + Size of incidental data

There's no hard-and-fast rule, but if you have to write a ton of data to the hard disk, and only some of it is actually related to the change you made, you have an inefficient pipeline.

## Potential Pitfalls

### Beware the Quick Fix

Treat the problem, not the symptom. Your process change may have unintended side effects.

For example, there is a general Configuration Management guideline that build and automation engineers will tell you: *Try to keep your local build environment as close as possible to, if not identical to, your build farm environment.* The pipeline installed on your local machine should be the same as it is on the build farm. The process by which the game and tool code is compiled should be identical on your local machine and the build farm. If those things diverge, you lose the ability to perform the "Find" step above effectively.

These days, distributed build systems are a very well-regarded optimization. They seem quite reasonable on the surface. However, those systems do break down periodically, so they may not be the best variable to introduce into a build farm, which needs reliability and repeatability above all. Also, they don't produce exactly the same executable that a standard compile and link would. That means that if you introduce them into a local desktop build, and you don't introduce them into your build farm, you have now broken the Configuration Management guideline mentioned above. There is the potential to run across some doozy of a bug found in the build produced by the build farm, and find out that you have trouble reproducing it because your executable is built differently.

So, use your best judgment here. There is no absolute right answer.

## **Beware Inserting Complex Process into Local Iteration**

In Part 1, we mentioned the complex check-in “gates” that a developer might have to work through to get his or her change into the game. In many cases, teams will apply the build and test farms to this problem. You can do this, but make sure that your build farm is ready to scale to be part of every engineer's workflow. You need to plan for the heavy usage that you'll undoubtedly experience near major project milestones. If you don't plan for peak usage, you will iterate the slowest when you want to iterate the fastest, because everyone will be checking in at the same time and efficiency will plummet.

## **Don't Let One Person Cripple the Team**

This one seems obvious. But this situation can occur a lot if you don't plan ahead. If it's not possible for someone to check in a mistake and fix it without taking down the team, you are asking for this problem. Everyone will have to wait while this person fixes his or her mistake.

## **WYSIWYG is Great, But Don't Try to Do Too Much**

The closer you get to WYSIWYG across your entire range of tools, pipeline and runtime combined, the better off you'll be. That is for sure.

However, some teams have introduced workflows that have created an untenable situation, resulting in loss of productivity rather than gain. For instance, in some cases, to shorten iteration time, teams have tried to recreate much of the look of the game in an environment outside the game, typically some sort of viewer application. They do this for what seems like a good reason: the time to load the game and advance to a place where the content can be seen is dreadful, especially during development. But there will be no way they will be able to keep up – the game will introduce a new feature, and that same feature will be missing from the viewer application. It will always lag, and will always be a source of frustration and pain. (See *Plan for Team-Wide Change* above)

Build your tools, pipeline, and engine so that they can work together to carry the change into the target environment as quickly as possible, in a form useful for the person making the change, following all the principles above. Each tool and each person on the team is a potential contributor to – or improvement of – the overall project iteration rate. By focusing on the factors that add time to your pipeline, you can increase your efficiency and more quickly and reliably produce a fun game.