Release This!

Tools for a smooth release cycle

*Frank Compagner, Guerrilla Games - Sony Computer Entertainment*

# Abstract

Over the past years we have developed a novel strategy for managing our release branches that, with the aid of some well thought out tools, has transformed our release process. This used to be a very chaotic affair, with long periods of confusion and uncertainty alternating with short intervals of blind panic. Most of our problems seemed to be caused by insufficient information on the part of the people supposedly in charge of the release, and an inability to effectively control change on the release branch. To fix both these problems we created *ReleaseManager*, a web-app that gave all developers and producers complete information on the state of the release branch, and *ReleaseTool,* which gave the build team complete control over what went into every deliverable.

# Background

Most of our products take several years to develop. The final year of production will typically see a good number (~10, last time) of, often overlapping, partial releases, culminating in a final shippable product after a long and intense release phase. A release branch was created for each of these deliverables, as usual. On previous projects, we ended up with most developers fixing bugs directly on the release branch (which would then be integrated back to the main branch). We found this to be lacking in a number of ways:

* Our projects have a head revision of about 300GB, with about 10GB of change per week on the main branch. Most developers have about 500GB worth of (very fast) local disk space. This makes switching branches a pretty expensive process, often taking most of a workday.
* As most developers ended up working directly on the release branch, and the final product is far too big and complicated to be tested properly before each submit, the release branch was often broken in subtle and entirely surprising ways.
* Producers in charge of a release had a lot of trouble determining which change had made it into what build on which branch. This resulted in mailing round questions about a lot of changelist numbers, doing many manual diffs to try to answer these inquiries, and a general air of confusion about the actual state of the release branches.

Because of this, we often found it easier to just let people continue working on the main branch, and only integrate their work to the release branch under complete control of the build team. At first this process was managed by manually keeping track of the status of all changes through mail and wiki pages, but this did not scale well. In fact, we found it almost impossible to maintain for more than a week or so, at which point it became very hard to figure out which change had made it onto which branch (especially problematic during times when we needed to support 4 simultaneous release branches).

Still, this allowed us to set up a release cycle as follows: at the beginning of the week, we would do a complete re-synchronization (or ‘copy up’) of the main branch to the release branch. The result of this then gets tested by our internal QA department, any problems found would be fixed on the main branch and integrated onto the release branch, where the fix would be verified. This normally would enable us to deliver a fairly high quality build at the end of every week, which would then go on to more stringent external testing. Several weeks before the final delivery of the release we would stop the *copy up*’s from the main branch, and spent the rest of the time ironing out the last remaining problems:

Intermediate Releases

Release branch

1 week

Main branch

Divergent change

Bug-fix

Final Release

Full copy

This worked reasonably well as long as we restricted ourselves to just fixing the highest priority bugs, but it did not scale well to more complex releases where we could tolerate less problems in the final product. It was simply not possible to keep track of the status of all needed changes, and the inevitable divergence between main and release branch meant that after a week or two we stated to see lots of conflicts in the integration results, often making it next to impossible to get certain changes on to the release branch.

# ReleaseManager

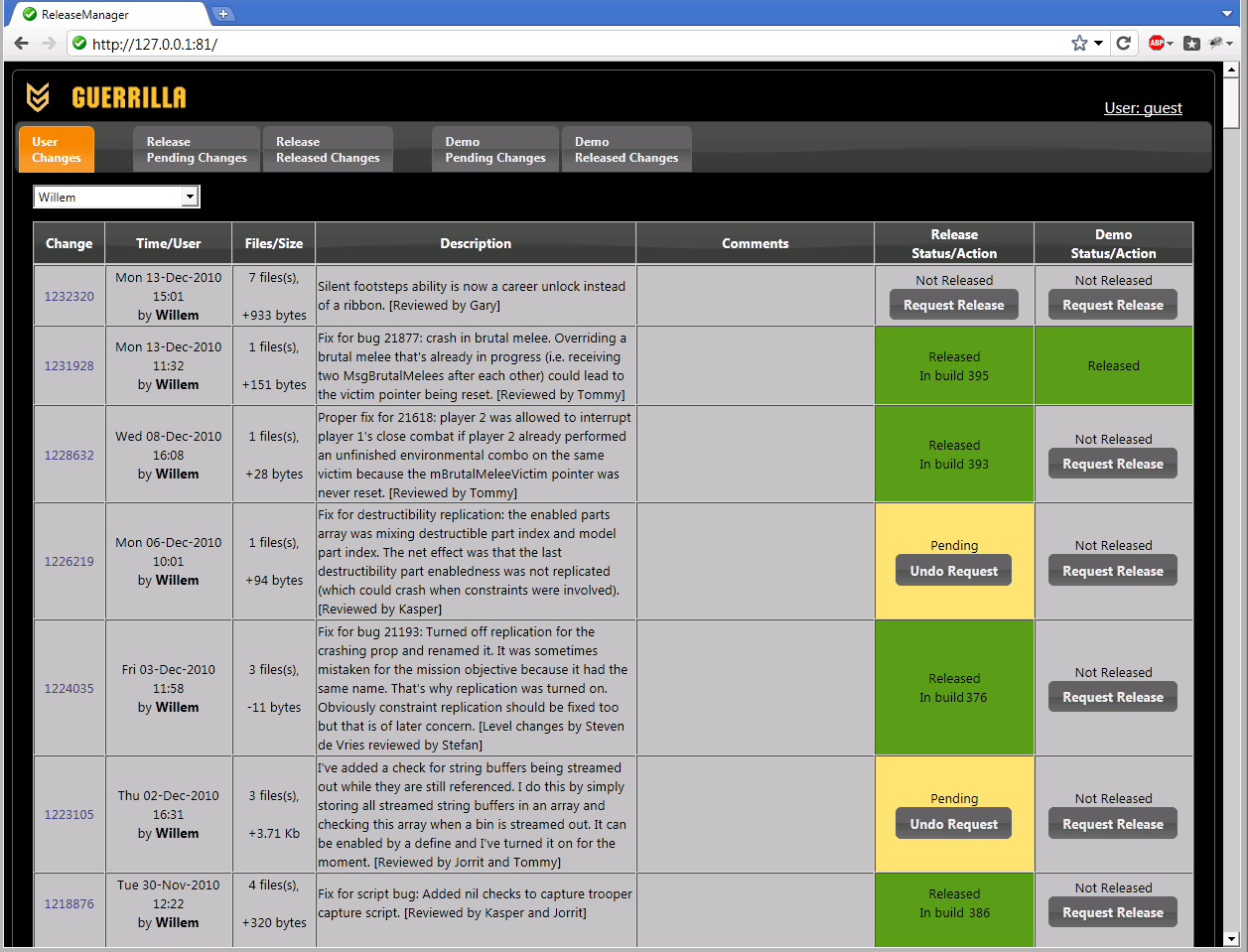
It became clear that if we wanted to continue in this way we would need better tools to manage the process. The central idea of our approach is that developers only ever need to interact with the main branch; responsibility for getting their changes onto any release branches is delegated to the build team. This simplifies things enormously for the developers, and everybody benefits from the clear separation of responsibilities. On the other hand, it does place a heavy burden on the build team, and it isn’t as easy any more for the developers to find out what happened to their changes (which branch are they on now, which build has them?).

As an important part of the problem was the lack of visibility of change status to everybody in the company, we quickly settled on a web-app that allowed all developers to easily see the status of all changes. The organizing principle of the *ReleaseManager* web-app is lists of changes submitted on the main branch. Any of these changes can be requested for release to any of a number of release branches. If the request is approved and the change is integrated, this will be clearly visible as a property of the original change on the main branch.

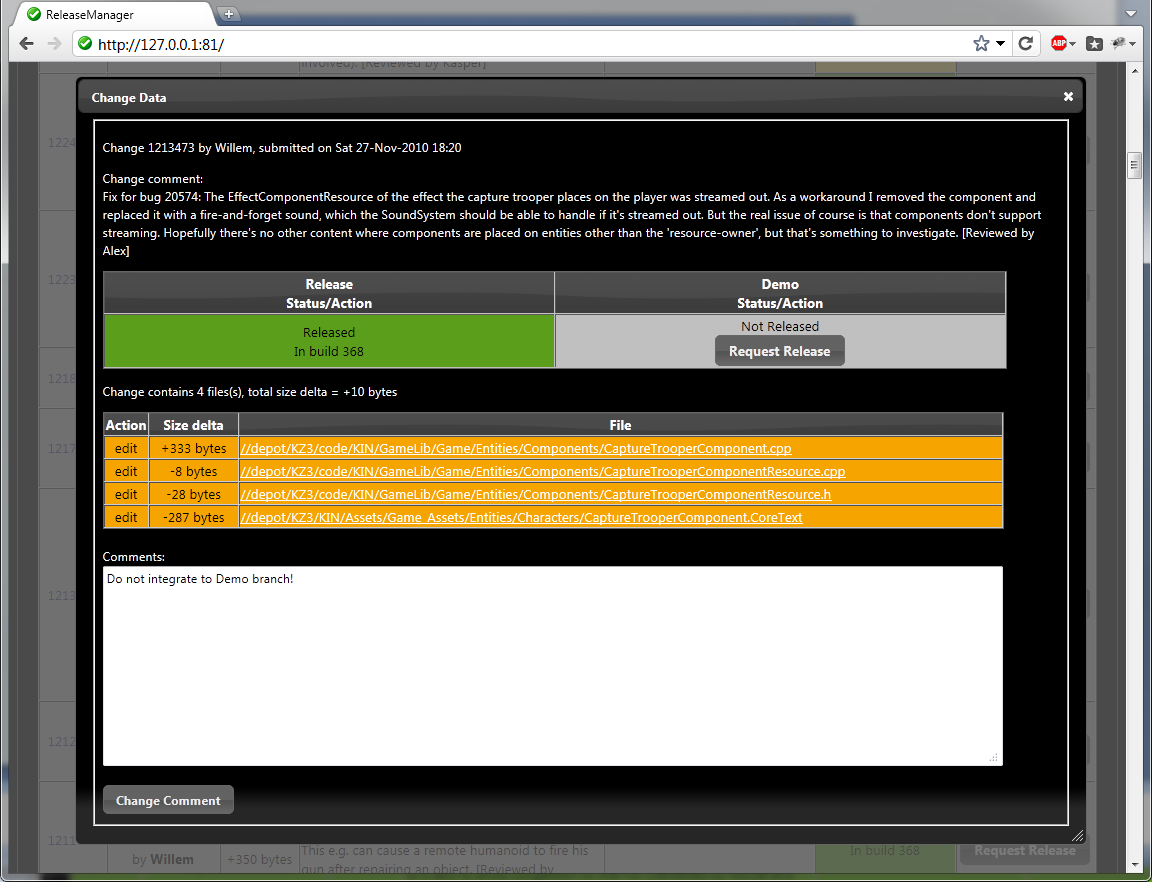
The ReleaseManager user interface is split in a number of tabs; the first tab with list all changes for a selected user, while for every active release branch there will be two tabs, with a list of requested (pending) changes, and a list of released changes.

## User changes

As can be seen in the screenshot below, the *User Changes* tab lists the changes of the selected user. For every change it lists the essential details, and for every active release branch it shows the status of the change on that branch. In the example below, the first change is not (yet) on any branch, the second one is released on both *Release* and *Demo* branches, though so-far only on the *Release* branch has a build (nr. 395) been made that includes this change. Integration of the fourth change to the *Release* branch (but not to the *Demo* branch) has been requested, and approval is currently pending. Note that the changes are presented in the order in which they were originally submitted on the main branch (with the oldest at the bottom), but that this does not necessarily correspond with the order in which they were integrated onto any release branch. Below, for instance, the 5th change is already present in build 376 of the *Release* branch, while the older 7th change only first appeared in build 386.

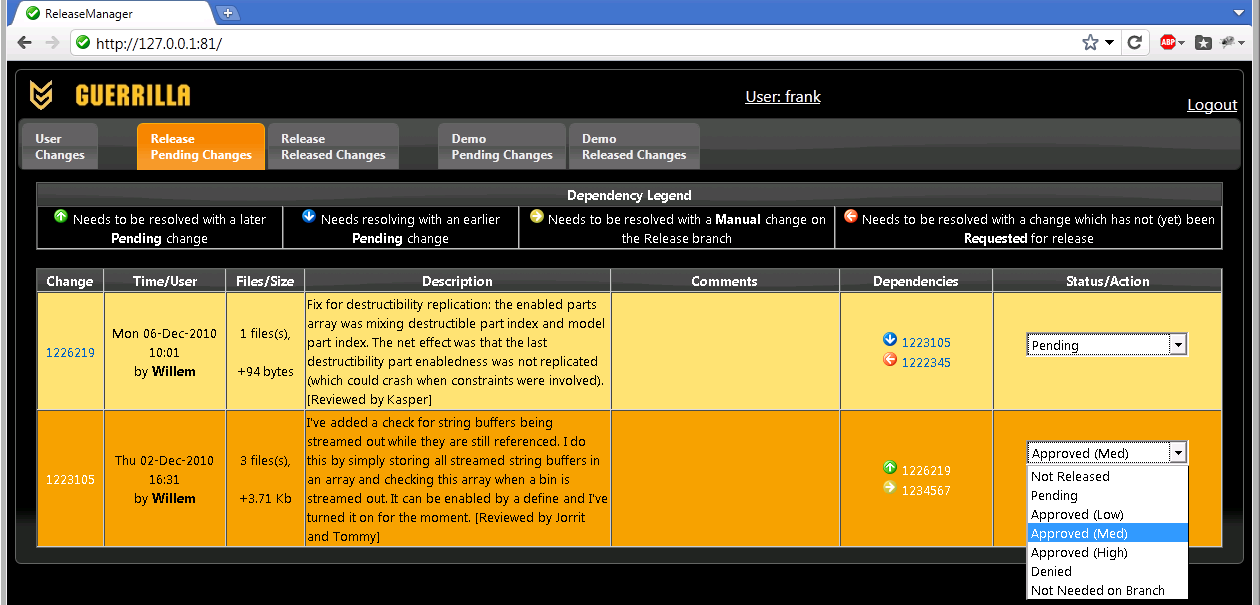


The changelist number in the first column is actually a hyperlink that opens up the change detail overlay below. Among other things, this contains a full list of all files in the change, each of which is a link to the p4web diff page that shows exactly what changed. It also has a comment field that can be used to discuss the change. Comments can be edited by anybody without logging in.



## Pending changes:

The pending changes tab gives, for every release branch, an overview of changes that have been requested for release, but have not yet been integrated. In this tab, members of the build team are allowed (after logging in) to approve or deny changes, and assign them a priority:



nav_up_green.pngThis tab will also show dependencies between changes. As soon as a change is requested for release, ReleaseManager will have a good look at the revision graphs for all files in that change. If it finds any previous revisions that haven’t yet been integrated, it will show a list of all changes that are needed to get the complete history of all files integrated onto the branch. Each of these changes is color coded to show where they are relative to the change in question. It is up to the build team to decide whether to try to integrate the full history of all files, but at least when they run into resolve conflicts, they have had a fair warning. Dependencies are categorized as follows:

Needs to be resolved with a later **Pending** change.

Needs to be resolved with an earlier **Pending** change.nav_down_blue.png

nav_right_yellow.pngNeeds to be resolved with a **Manual** change on the Release branch.

Needs to be resolved with a change which has not (yet) been **Requested** for releasenav_left_red.png.

The arrows point into the direction of the change in the ReleaseManager interface. Red changes will be found in the user changes tab on the left, yellow (*Manual*) changes will be on the Released changes tab (see below). And green and blue changes will be in the pending list, either below or above the current change. Below is an example of a revision graph of a single file with the location of each type of dependency shown.

Pending changes consideration

Manual change consideration

Integrated change

Change under consideration

The set of dependencies for the entire changelist is simply the union of the dependencies of all the file revisions that make up the change.

## Released changes:

Finally, the released changes tab shows all changes that were integrated onto the release branch, in the order that they were submitted (oldest at the bottom). It also lists any builds that were made from the release branch, making it very easy to see exactly what went into each build. ‘Manual’ changes that did not originate as a change on main, but were made directly on the release branch are shown in a distinguishing color. Although the vast majority of changes started out as a change on main, manual changes like this were still possible and necessary to properly tweak and polish the final deliverables. We also provided dead simple bug tracking integration; any number in description or comment text that looked like it might be a bug number (that is, anything between 1 and 30000) was changed into a link to the corresponding devtrack page. Obviously, a proper jobs system would be better, but this still worked pretty well. Similarly, any number over 1.000.000 that matched a known change was adorned with a link to the corresponding change details overlay.



## Implementation:

The web-app itself was made in Python, using the light-weight CherryPy framework, p4python for the Perforce backend, JQueryUI to provide a friendly user interface and a simple python shelve as the database. The whole thing is fairly simple at about 1500 lines of python and some 300 lines of Javascript; the initial version was put together in under two weeks. After it was found to work well, a considerable amount of work was still needed to add support for build and dependency information and to make the tool scale to over 30.000 changes and half a year of history.

The ReleaseManager application consists of a background thread and any number of UI threads that serve the actual pages and respond to user input. The background thread performs three essential steps::

1. Checks if new builds have been done on the release branches. If so, the build number, changelist number and time/date of the build are recorded.
2. Checks to see if new changes have been submitted on any of the release branches. If so it tries to determine the corresponding original change on the main branch (see below for details). The status of newly found changes is set to ‘*Released*’ or ‘*Manual*’.
3. The dependencies of any changes found in step2 (and of any changes that were depending on those changes) need to be updated.

Meanwhile, the UI threads will query p4 whenever the user changes tab is selected, making sure that that list is always up to date. That makes it possible to submit, open ReleaseManager and request the change for release immediately. The UI threads also take care of changing the change status (to ‘*Pending*’, ‘*Approved*’, ‘*Denied*’, etc.) and will also update the change dependencies as needed.

An important implementation detail here is that the ReleaseTool (see next section) will use a specific change description for all the integrated changelists it creates. These will typically be something like this:

Integrate //depot/KZ3/...@1235471 to //depot/KZ3-Release/...

Original message:

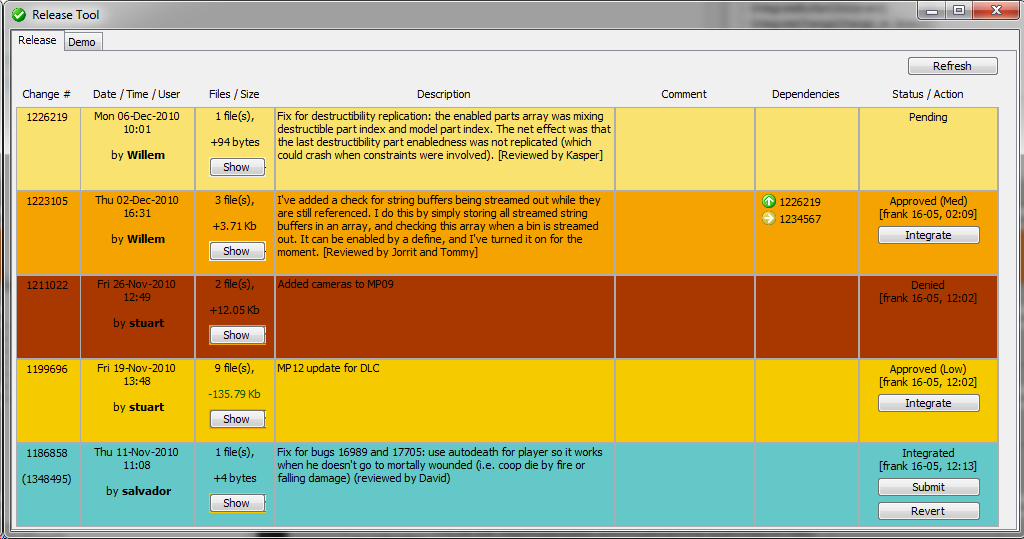
Change 1235471 by frank@frank64 On Thu 16-Dec-2010 15:26

Possible fix for bug 21968.

It is these change lists comments that allow ReleaseManager to figure out which changes have been integrated onto which release branch. If a change on the release branch does not conform to this template, it must have been a change made outside of the ReleaseTool, and there will be no matching change on the main branch. Such changes are flagged as ‘*Manual*’. While this may sound rather brittle, we have found it to be absolutely no problem in practice.

## ReleaseTool:

So far we have only looked at the web-app that is used to communicate changelist status and other meta-information. While this is certainly the central tool in this approach, we also need something to do the actual integrations. To handle this we created another tool, called *ReleaseTool*. It too was made in Python, using the wxPython GUI lib, and is used exclusively by the build team. It talks to ReleaseManager to get a list of pending changes for the selected release branch. It shows these changes as a list (oldest change at the bottom, as on the pending changes tab), and for each approved change it offers an “Integrate” button. This button does exactly that; it will integrate the change in question to the release branch, followed by a safe ‘resolve –as’ (or, in the case of unmergeable binary files, resolve –at). If this resolves all files in the change, it is ready to be submitted and a button labeled ‘Submit’ will appear. If it is still necessary to resolve some files, there will only be a ‘Revert’ button, and the build engineer is expected to go into P4V to see what all the fuss is about.



A commandline version of this tool (called *AutoReleaseTool)* also exists; it will try to batch integrate a number of changes in one go. This was created after we had considerable experience with the interactive tool, and we had gained enough confidence in our changelist selection that we let the AutoReleaseTool do a resolve –a (i.e., accept anything that can be merged without conflicts automatically). It will essentially plough through the list of approved changes and will only stop at the first resolve conflict. This worked surprisingly well and did not nearly cause as much problems as we initially feared.

# Conclusion

While the original reasons for letting people continue to work on the main branch were essentially negative (getting them onto the release branch was too hard), we found that accommodating this with the tools described above gave us unprecedented control over our release process. We gained complete control over the order and timing of changes on the release branch. This allowed us, for instance, to test certain more complex features in isolation on the relatively stable release branch, without much interference from other unrelated changes. It also allowed us to sometimes postpone risky changes until just after an intermediate deliverable, thereby maximizing testing time. In the end the amount of control over the entire release cycle we regained in this way is the main reason we have really grown to like this process.

Of course, there are a number of obvious downsides to this approach. First and foremost is the fact that nobody in the entire company has the actual release branch on their machine; it exists solely on our build servers and as a finished build at the QA department. This means users cannot easily see the actual state of the release branch, so that reproduction of bugs is potentially a problem. In practice this did not present a major problem, partly because through the entire procedure we managed to limit divergence between main and release branches, and partly through a very thorough QA effort. For emergencies we did have one or two users that had direct access to the Release branch.

Also, as the order in which changes are integrated to the release branch is sometimes completely different from the order in which they were originally submitted on main, there is much potential for complex resolve conflicts. We definitely found that if there is any major development still done concurrent with the release, this is much better done on a separate dev branch, otherwise it will start to interfere with normal bug-fixes very quickly.

In essence, the ReleaseManager creates a fiction about the state of the release branch. This fiction is very useful, as long as it doesn’t stray too far from fact. It is up to the build team to make sure that this doesn’t happen; they need to be very careful in resolving the inevitable conflicts, and need to stay on top of all release details to make this work.

But when it works, it *really* works. We have never had a release as smooth as this one, and we are pretty sure that ReleaseManager played a big role in that. In essence ReleaseManager gave us:

* Complete and accessible information for all users.
* Reduced confusion, improved communication between developers and build team.
* Awareness of branch and change status boosted, confidence in builds improved.
* Better decisions could be made under pressure.
* Control was regained.

This process makes the build team responsible for integrating all changes to the Release branch; using ReleaseTool they were able to do so efficiently and sensibly.