GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Presto Studios' *Star Trek: Hidden Evil*

By Michael Saladino

Change... it is really the only thing you can count on in this industry. And if you cannot adapt, you are pushed out of the way. That was the situation at Presto Studios years ago as real-time 3D engines slammed onto the gaming market. In response, the decision was made that Presto would follow this trend and create its own 3D engine. For those that do not have any knowledge of Presto Studios, it is most widely known for the creation of the *Journeyman Project* series of graphic adventure games, ala *Myst*. Beautifully prerendered backgrounds, puzzle solving, and Director programming is what built the company. But now the founders had decided to push aside all that they knew in an attempt to make their mark in real-time 3D. So a group of programmers were hired with the experience and knowledge to create a new engine from scratch. The first released product from this effort, *Star Trek: Hidden Evil*, is a clever blend between where Presto was and where it wants to be.

We entered into a partnership with Activision in order to create the first game with their new *Star Trek* license which was acquired from Paramount. Due to our experience with prerendered adventure games, we were given the task of creating a prerendered action/adventure title. They wanted a mission based game with step-by-step hand holding instructions to appeal to casual gamers. They wanted it to be small so that casual gamers would not be intimidated. They wanted it to have beautiful backgrounds with real-time characters moving through out the scenes with per-pixel depth sorting. But most importantly, they needed it to ship in one year so as to hit Christmas 1999 and thereby fit into Activision's grand marketing view for it's entire line of Star Trek games. In the end, I believe we succeeded in creating exactly the game Activision and Paramount wanted even though it was a rough road to our final goal.

Our development team consisted of two very different groups of people. The programming team was mostly new to the company with serious experience in doing real-time 3D games. I, the lead programmer of *Star Trek,* came from Volition, Inc. of *Descent* and *FreeSpace* fame. The chief technology officer of the company has fifteen years of 3D graphic pipeline experience. Our artificial intelligence programmer has a Ph.D. in his field. We were the programming core that helped ease the company into its first real-time game.

The production team mostly came from the other side of the gaming world where design and rendering quality are top priorities. They mostly came from the teams that created the *Journeyman* projects for Presto years ago. Luckily, there were a few new artists with real-time 3D experience that helped bridge the gap between the two camps.

The production was divided into these two sections: the real-time group and the prerendered group. The real-time group was responsible for creating character and object models, animation states, cut scene animations, collision geometry, and gameplay logic. These tasks were done through 3DSMax along with a suite of in-house development plugins that were used to export worlds, objects, animations, and gameplay logic. The prerendered group was responsible for designing the scenes, creating background images, and prerendered cut scenes. This was done in Electric Image with Photoshop touchups.

We Delivered What We Promised

Even though details of our game changed wildly between initial conception and final execution, the fundamental goal was thankfully consistent: create a mission based, action/adventure game for casual gamers. And our final result hit this target extremely well. One of the major factors in our success was our push to always error on the side of "too easy." A piece of market research out of Activision stated that of people that own game-capable PCs and enjoy watching *Star Trek*, only a small fragment had ever purchased a *Star Trek* computer game. Number one reason as to why they had never bought one: "They look too difficult." That was our market. We wanted to give casual gamers the opportunity to be in an episode of *Star Trek*. We wanted them to be able to talk to Picard and Data, walk around and examine objects with their tricorder, and shoot things with their phaser. That is Star Trek, and therefore, we decided that would be entertaining for the fans.

***Star Trek: Hidden Evil* (left) represents a new direction in game development for Presto Studios (right)**

In order to appeal to novice gamers, we chose to keep the action side of the game simple. Our market is not looking for death matches with fast paced projectile warfare. Instead, we simplified the concept. Auto-aiming was a large factor. Since the game has a fixed position, 3rd person camera, having to control your character in order to aim at another would simply be too difficult; therefore, a simple cone test determines if you are facing enough towards an object for your character to automatically lock his aim onto the position. In order to promote longer life, we kept the damage done by enemies very low, while keeping the player's phaser strength high. We tried to remove as much "twitch" combat as possible and rely more on the entertainment derived from the sudden shock of a new enemy jumping (or flying) into camera view and the following pleasure received from watching them fall from a phaser blast.

Puzzles were the other main gameplay aspect that we needed to balance towards the casual gamer. We kept the puzzles challenging and with good rewards to propel the players forward. Finding new breathtaking areas to explore and watching real-time objects react to things you do, these make excellent rewards for puzzle solutions. In ten missions, there are five entirely different locations to explore from the Enterprise to an ancient underground civilization. Each with its own feel and gameplay. This gives casual gamers hours of entertainment from just exploring the world.

We Delivered When We Promised

A big success that I'm sure our publisher Activision would agree with was our completion of the game on time. In September of 1998, we entered into a contract to deliver a gold master CD to them in mid-October 1999 and we hit that mark with the product they wanted. This almost guarantees the game's success from a financial standpoint since it will make Christmas. Software retailers will be selling *Star Trek: Hidden Evil* with Jean Luc Picard on the box. Holiday shoppers will recognize the face and the name, and that will most likely get them to pick up the box. From here, it's an easy $29.95 to purchase what they know as opposed to scores of other more expensive games that are unknown to the casual gamer.

By hitting our gold master deadline, we stayed within our budget. The game succeeded in being a small, low cost adventure that will not overwhelm a first time gamer. Whenever a game slips, costs begin to rise due to salaries and resources are drained from other projects. Your profit margin is eaten into quickly during the time past your deadline so of course finishing on time is extremely important for the financial wellbeing of your company.

Being released on time also saved money for Activision. Every year, a major publisher releases a half dozen or more titles during the Christmas season. By getting our project finished in October, we freed up valuable resources in their QA department. By completing the product within the timetable that Activision had been planning on for the last year, we did not force any massive rescheduling of their advertising plans. Shipping a game on time is good for everyone involved, from the developers, to the publishers, to the consumers.

The Introduction of New Blood

Every now and then, a company makes a decision that just happens to turn everything around. Or at the very least, makes the impossible possible again. This is how I would describe our hiring of two interns, Andy Schatz and Narayan Brooks. Half way through the project we realized that we did not have the manpower to finish our level instancing. (Level instancing is how we refer to level construction, setting up doors, placing enemies, etc...) Now while this might sound odd to those that come from companies creating *Quake* clones, remember that Presto Studios has been making prerendered *Myst* style games for years. Until we ventured into real-time 3D, creating levels was the job of artists who worked on scenes with millions of polygons. Creating low level geometry and placing door logic requires different, more engine specific skills. So we hired two people who were working towards their degrees in computer science to help us lay down the logic needed to make our 3D worlds come to life.

Since they both were given the very specific task of using our tools to make the worlds come to life, they were able to experiment. This was the first time that anyone really sat down and explored what our tools could do. Add this to the fact that they both seemed to love making the game, it was a perfect solution to multiple problems. All of a sudden we had people who wanted to not just construct what they were told but were actually forging ahead with new ideas. Creating scenarios that the level designers had never considered. Their excitement soon caught on to others on the team and really helped propel the project towards its final goal.

Making The Hard Decisions

One of the hardest decisions any game company can make is when to cut the dream. Can those in charge decide before it's too late that the game as designed is simply too big? Luckily, the answer to this question was "yes" at Presto Studios. The initial game design was a

mammoth undertaking especially when you consider the one year timeline. As we watched ourselves fall behind early with milestones, we knew that unless we changed the project significantly we would completely miss our gold master date and Christmas with it. It was decided that cuts had to be made.

The entire team sat down for nearly a week doing a complete walk through of the game. Discussing every level, every scene, every event, and making sure that everyone involved understood what needed to be done. Many items were axed immediately as people heard, sometimes for the first time, about a situation that needed to take place for the story to progress during the game. Other ideas were stripped down and rebuilt in order to use the engine technology better. By the time we were done, an entire level had been removed and all others had been significantly reduced. We sugar coated certain cuts with the phrase, "If we have time at the end", but as everyone knows that means it's dead. No one in the history of game production has ever had time at the end.

And this was by no means the end of the cutting. We still had five more months of cuts to come. Over the rest of the project, the occasional snag would develop. Perhaps the engine couldn't perform some task the way the designer had envisioned. Perhaps the schedule was getting too tight and we just did not have the manpower to animate a given cut scene. Across the board, if the feature was threatening our release date, it was cut. Nothing mattered more than shipping on time and we all agreed that shipping a small solid game was better than shipping a large unstable one. Plus, the Activision play testers never noticed that anything had been cut. The missing sections are only visible to those of us that knew they were once suppose to be there.

The Rendering Engine

Some say I am a programmer. Well, in order to support that idea, I guess I should talk about a programming task that went right in a big way... the rendering engine. The initial concept of the engine was simple enough, prerendered backgrounds and real-time objects mixed with per-pixel depth sorting. We already had a real-time rendering engine which was used for creating *Beneath*, a real-time 3D action/adventure game which was put on extended hold. However, the addition of the prerendered backgrounds proved to involve far more code than most suspected at first.

Examples of the combination of pre-rendered and realtime 3D components of *Star Trek: Hidden Evil*

I chose a depth buffering algorithm in order to handle the per-pixel sorting. Electric Image could export it's z-buffers which I then converted to our own custom depth format. I had to write an entirely new renderer since the *Beneath* renderer used a span based hidden surface removal algorithm. This required rewriting all the pipelines including transparency, translucency, shaded, etc... A camera subsystem was created which could handle loading backgrounds when needed. I wrote a dirty region update system based on scanlines which could rebuild damaged parts of the background and depth buffer as real-time objects passed over them. (Add in the translucent heads up display which overlaid both the backgrounds and the real time geometry and we had quite an interesting task in optimizing the updating.)

However, the most interesting part of the code in my opinion was our solution for hardware rendering. We needed to follow the same concept with prebuilt depth buffers that could be blit copied over to the card and then have the card render triangles correctly into them. My initial thought was that I would need to reverse engineer every card on the market in order to determine how they handled depth buffers. Would this card use some floating point format, would this card use w-buffering, would this card use greater numbers as the object got closer to the camera? The result of my experiments surprised me. Of all the depth buffering cards we tested, none of them cared what information you stored in their depth buffers as long as you sent the appropriate data through the triangle renderer. My final solution has all cards being loaded with the same 1/z fixed point depth data. As long as the values I blitted into the depth buffer matched up with the values I sent to the card in the triangle vertex block, everything worked. In other words, cards don't perform any magic when you send z values through Direct3D or Glide. They simply interpolate the values across the polygon (in a perspective corrected fashion) and store the results in the depth buffer. Writing a value straight into the card's depth buffer is like drawing a single pixel triangle at that screen location at whatever depth you send.

It's All About The Tools

In the real-time 3D arena, it's all about the tools. What's the point of a brilliant rendering engine that is faster and more beautiful than everything else on the market if your own production team cannot get anything done. It'll be a great demo, but not a game. The first generation of tools at Presto was practically a step-by-step course in how to not create tools. The incremental design ate away at all sides until the whole structure threatened to fall in on itself. The code for the tools was bad in that they never were crash proof. And in our mad rush to get in more features, we never seemed to be able to spend the time to fix these sporadic problems. So bad code was covered up with more bad code and the situation grew worse as time progressed. All of the programmers at the office, except for the one that wrote the tools,

viewed them as some sort of black magic that has been written in another language. The production team was left with little help when something did not work. Only one programmer could help them and he never had time. It was not a good solution. You must never allow one programmer to be so critical. Multiple programmers should be able to handle any given system.

One of the best things a tool can do for both the production team and the programming team is to explain with errors and warnings when something goes wrong. Unfortunately, our tools programmer did not quite see the importance of this. Serious flaws in the data being exported would go unannounced while other warnings were actually used by artists as signs that they had exported something successfully. Giving incorrect warnings is worse than giving no warnings because now you are training the user to ignore them. Proper error checking would have saved everyone weeks of time spent debugging broken game logic that ended up being an improperly setup item in 3DSMax that the tools should have reported.

To make the situation worse for the production team, the interface was designed by the programmer as he went. Once again, incremental design reared it's ugly head and bit off a large chunk of our productivity. And the fact that a left brained programmer was creating a graphic interface for right brained artists is a mistake that I'm sure most game makers have seen at one time or another. It just does not work. Without any good pipeline for requests from the production team to get the programmer, the tools continued to become unusable. In one of the tools, a button marked "Destroy All Data" sits directly next to the "OK" button. With no "Are You Sure" check on either, this perfectly personifies our first generation tools.

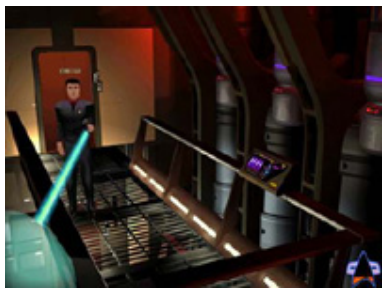
Understanding The Big Picture

After digging in deep to the programming side of the tools, I now feel justified in pointing the spyglass to the other side of the team. Production had every right to hate the tools but this is not a good reason for not using the tools. Many members of the production team lacked any serious knowledge of the engine or the tools used to create worlds. This included people who were designing gameplay on paper without any real understanding as to the construction pipeline for their ideas. However, both of the interns we hired halfway through the project became quite proficient at navigating the mine field that was our tool suite. How is it that people who were around for years while the engine was being first developed understood so little, while our newest people were able to grasp the concepts in a matter of weeks. I believe it comes down to who wants to make games and who wants to make art.

But don't we all want to make games? That is why were are all in the industry, right? Well, yes and no. Some of us study all sides of games. From script writing, to programming, from interface design, to production management, I've gotten my hands dirty in everything at one time or another. I don't do this because I feel that I must be an expert in all these skills, but I do feel like I need to be proficient enough in these skills in order to work intelligently with everyone on the team. While one purpose cogs can work effectively in large development houses, small teams often require members with much more generalized skills. Everyone should understand on at least a fundamental level how what they do will affect other team members and in turn the final product. And being able to do this requires basic problem solving skills. If you do not enjoy fixing problems then you most likely will always be frustrated making games.

Conceptual Design vs. Level Construction

Possibly our worst decision in making Star Trek was our separation of level design and level construction. The decision came from the company's experiences with it's previous graphic adventure products. In this design setting, having one artist create the visual and mechanical concept of the level and another model the level from the drawings is a perfectly viable pipeline. However, in a game with real-time navigation and combat, trying to design something for beauty will not necessarily get you something that is entertaining. When our two interns started experimenting with our tools and engine, they developed scenarios that our conceptual designers never could. Scenarios that were not only exciting but were easily implemented due to the innate nature of our technology. Knowing the tools is one of the most important skills any person involved in level design can have.



Camera angles had to be chosen not just for their aesthetic qualities but also for their effectiveness in combat.

Our product pipeline was made inefficient due to our separation between design and construction. By halfway through the project, we had created entire levels without ever setting up combat scenarios or even attempting to walk the hero through the environments. Our camera shots were chosen for their aesthetic qualities and not for their effectiveness during action scenes. And whenever a problem was found in the placement of a crate or a camera, it required getting into prerendered Electric Image files, making changes, rerendering, and then trying out the new scene often with the result being that we didn't change it enough or we changed it too much. The pipeline from concept to execution

was simply too long. Prototyping our levels with visible collision geometry would have been a much shorter pipeline involving only changing a small file in 3DSMax and exporting it to the engine. One person could have experiment with dozens of setups every day instead of one setup every couple of days.

Technical Design As You Go

Poor design is not just a problem for artists. Programmers need to design systems before incorporating them into a project especially when there are other programmers. We suffered from the no-design mentality quite a bit on the Star Trek project. Since we were essentially building the game and the engine at the same time, we were constantly running into time conflicts where we needed a new feature right now in order to meet a deadline next week. This resulted in what is often the standard in game programming: design as you go. Our basic game play elements such as path followers and rotators were built up piece by piece often by different programmers, none of which had a full understanding of how the system would or should behave in relation to the rest of the code base. Ambiguities were created, partially functional code was activated, nothing good came from it. Except for our current dedication to make sure that this never happens again. We have now incorporated design meetings and multiple stage code reviews into our daily routine. The results have already been astounding.

One interesting example of the opposite problem arose in our custom interface system: we had too much design. Our interface system is the library of code that handles our front end interface and our in-game heads up display. The code was written by a programmer with extensive experience in C++ object oriented design and loved to use it. He spent many weeks designing and crafting an interface system which was extremely... well... designed. Private data fields were placed into private headers. The "const" operator was used everywhere. Void pointers hiding internal data structures made debugging a nightmare. The entire system was designed with the idea of keeping "the client" from changing system internals or even from being able to understand them. And while that style of programming has its place in a production team with 200 programmers whose goal is to sell a simulation library, it has a tendency to slow down the work of three programmers working together on closely spaced milestones. In other words, for the other programmers on the team the smallest change to the system required hours of exploring and debugging and usually resulted in them throwing up their hands, cursing loudly, and hacking in a variable called *yet_another_interface_hack_for_inventory_menu_hot_keys*. Not a pretty sight. There are extremes at both ends of the design spectrum and either one can seriously damage overall productivity. Make sure that you spend time designing what is important.

Cutting A Story Based Game

When creating a game based on story, you will always run into the following problem throughout the project. How can you cut that, now the story doesn't make sense. The first problem was that we had to cut so much from the game. There were two primary reasons why we had to cut so much. First, we were running out of time with such a short production schedule and we had to finish in time for Christmas. However, beyond that, many cuts were made simply because the story was too static. In a game, the player is in control of the small details. When a camera cuts, how a combat unfolds, these are details that are mostly out of the hands of the level designer. This is actually the whole reason why people play computer games. They want to control these things. If they didn't, they would go to the movies and save forty dollars. Therefore, designing a scenario in a game that spells out every single detail is never going to work. If you hear a pitch for a game scene and it's filled with sentences like, "And then the hero pushes the killer down the stairs, runs to the desk, pulls out the knife, but when he returns the killer is no longer at the bottom of the stairs."; that's not a game, that's a movie. (And seems like a bad one at that.)

In a game, you give the player options and it's up to them to select what they want to do. If you want the player to be able to push someone, you have to work that control into their standard control interface. It needs to be something that can be done repeatedly. There is no point is spending the time designing and building something that the player can only do once. This means that this action has to stand the test of repetition. Can it be done over and over again and still be fun? Do not try to script out combat situations except for the initial trigger. Once the bullets (or phaser blasts) start flying it's up to the player and the game logic to determine what happens next. You cannot assume that the hero will run here or there. Who's to say that he'll run at all. Maybe he'll walk slowly away from the action. Maybe he'll just stand there. It's up to the player and that's what makes games unique. It's one of the only mass market entertainment formats where the buyer can truly interact with the product.

After removing many scenes from the game, we needed to repair the holes left in the story. Our solution for many of these cuts was to create a prescripted cut scene. Some of them showed basic actions or events but most fell into the category of "data dump." These are scenes that obviously exist for no purpose other than to convey information to the player. The result in Star Trek was one or two minute dialogues with characters standing around talking, sometimes over communicators which really doesn't create an energetic scene. Our use of dialogue to convey information breaks one of the fundamental rules of visual entertainment, convey information with images. Instead of watching someone tell you about a level, why not listen to the voice over while watching a fly-by of the level on your tricorder.

In Conclusion

The current state here at Presto Studios is one of excitement. No one is really too concerned over the amount that went right or even the amount that went wrong on *Star Trek*, the final result is that we finished a good product on time. However, I am thrilled over the amount that everyone here has learned. Our initial steps towards the "next big thing" have been quite promising. Our production team is now more focused and organized towards real-time engine development. Our coding staff is working intelligently towards next generation tools and engine technology. Those people that did not share in the vision have left for jobs that better suit them and we wish them luck. I have been through many projects in my career and many of them have never seen the light of day. But never in my experience have I ever seen a group of people learn so much so quickly from their mistakes. I hope that this look into our project, *Star Trek : Hidden Evil*, has allowed you to see

some possible flaws in your own team. And maybe give you an idea or two on how to reorganize and move towards a more cohesive team environment.

Project Data

The *Star Trek: Hidden Evil* Team: Left to right

***Top Row*: Lars Liden, Ph.D. - AI Programmer Jamey Scott - Composer / Sound Designer Tim Tembreull - Producer Eric Dallaire - Writer / Game Designer Derek Becker - Pre-Rendered Artist Eli Enigenburg - Technical Director / Animator Phil Saunders - Design Slave Mike Brown - Animator**

***Middle Row*: Francis Tsai - Conceptual Designer Sean Keegan - Modeler Victor Navone - Creative Director Michael Saladino - Lead Programmer Narayan Brooks - Level Builder**

***Bottom Row*: Dan Gregoire - Lead Pre-Rendered Artist Casey Steffen - World / Level Builder Keith Gurganus - Programmer Max Elliott - Senior Programmer / CTO**

***Not Shown*: Michel Kripalani - Executive Producer Andy Schatz - Level Builder Raymond Wong - Pre-Rendered Artist Steve Kim - Animator James Rochelle - Texture Artist Scott Benefield - Creature Design**

Star Trek: Hidden Evil

Developer: Presto Studios
5414 Oberlin Drive, Suite 200
San Diego, CA 92121

Publisher: Activision

Team:
Tim Tembreull - Producer
Eric Dallaire - Writer / Game Designer
Victor Navone - Creative Director
Michel Kripalani - Executive Producer
Michael Saladino - Lead Programmer
Max Elliott - Senior Programmer / CTO
Lars Liden, Ph.D. - Programmer / AI Specialist
Keith Gurganus - Programmer
Casey Steffen - World / Level Builder
Andy Schatz - Level Builder
Narayan Brooks - Level Builder

Francis Tsai - Conceptual Designer
Dan Gregoire - Lead Pre-Rendered Artist
Derek Becker - Pre-Rendered Artist
Raymond Wong - Pre-Rendered Artist
Eli Enigenburg - Technical Director / Animator
Mike Brown - Animator
Steve Kim - Animator
Sean Keegan - Modeler
James Rochelle - Texture Artist
Scott Benefield - Creature Design
Jamey Scott - Composer / Sound Designer

Development Time: One year

Release Date: November 16th, 1999

Minimum Platform: PC Pentium 200 MHz 32 Megabytes of RAM 3D hardware acceleration optional

Hardware Used to Develop: PII ~400Mhz - Programming,
PII ~400Mhz - Real-time art, Macintosh G3 - Prerendered art, Six PowerMacs - Render farm

Software Used to Develop: Microsoft's Visual C++ Developer 6.0, Microsoft's SourceSafe 6.0, NuMega's Boundschecker 6.0, Intel's Vtune 4.0, Kinematic's 3DSMax 2.5, Play's Electric Image 2.9, AutoDesys' Form Z 2.9.5, Adobe's Photoshop 5.0, After Effects 3.1, and Premiere, Terran Media Cleaner 4.0

3rd Party Libraries: DirectX 6.0 (using DirectDraw, Direct3D, DirectInput) 3Dfx Glide 2.0 Miles Sound System 5.0q QuickTime 4.0

Michael Saladino was lead programmer on Presto Studio's *Star Trek: Hidden Evil*.

Return to the full version of this article