GAMASUTRA
The Art & Business of Making Games

PRINT

Student Postmortem: 6mSoft's *Romeo and Juliet*

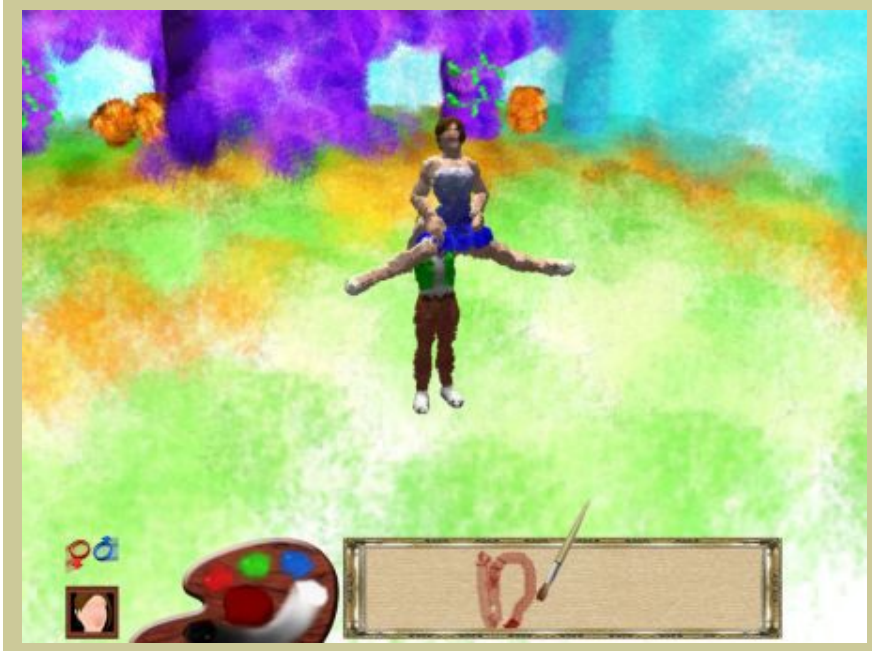
By Brian Gilman

As part of the Bachelor's completion course at Full Sail game design school, every student has to take a final project course. The course is five months long, with two months of pre-production and three months of development time. During the course, the class is structured as a studio, and acts and makes decisions as a studio. In the first month of pre-production, the class decides what game concepts it is going to pursue. To facilitate this, the class divides into pairs to develop a game concept, which will then be presented to the class. This is where the seed of *Romeo and Juliet* took root.

At the 2004 Game Developers Conference, there was a game developer's challenge where three veteran game designers (Raph Koster, Warren Spector, and Will Wright) were tasked to design a "love story" game. My partner and myself felt that this would be an interesting challenge, and on top of that, it was something that hadn't been done yet, so we proceeded to draw a rough sketch of what the game would be like. This rough sketch was presented to the class, and was chosen as one of the projects that the studio would develop. We split into teams and started designing our games.

The first thing we decided was that the love story would be told through ballet. Ballet is unique as an art form in that it is a highly emotional form of expression, yet, uses no language. We felt that this was advantageous, as having to deal with language would have been a gigantic black hole.

Interaction with video games is currently done at an almost entirely rational level. The player may react to a game emotionally, but the game will never know about it, and thus, never respond to it. We wanted to change this, and have the player interact with the game solely through his own emotions. To do this, we decided on having the player express himself by painting with a paintbrush. The player would have three colors to choose from (red, green, and blue), as well as being able to select brightness. Each color would be dualistic in that it could express two emotions, one positive, and one negative. For instance, red could express either love or hate, with bright red being love, and dark red being hate.



Painterly rendering in *Romeo and Juliet*

We also wanted to give the player as much feedback as possible, in multiple forms. This included interactive music, painterly rendering, which is a method of rendering using brushstrokes, choreography and animation. All of which would change depending on the current emotional state of the relationship.

With our plate thoroughly full, and on a very short timeline, we journeyed into the unknown.

What Went Right

High Concept. Having a creative, and an extremely different high concept for our game helped our project in a number of ways. The initial blank stares we got from people we introduced the project to, quickly turned to excitement. We were known as "the ballet guys" around our building. Everyone we talked to seemed genuinely intrigued about the project. This excitement gave us momentum coming out of the starting gate; when we were tackling difficult, to the point of demoralizing, design issues. If it hadn't been for the excitement that had built around the project, there is no doubt in my mind that the project would have turned out for the worse.

The high concept also enriched the learning experience while we were creating the game. Unlike previous projects at Full Sail we had worked on, there was no existing game that we could use as blueprint. This forced us to be creative to a very high degree during the design process, carefully thinking through what we were designing, versus cherry picking features from games of genre X. It also forced us to venture outside the comfortable domain of software engineering. In order to complete our design, we had to be truly cross-disciplinary, and be psychologists, storytellers, ballerinas, choreographers, and directors, all at once.

The high concept also appealed to female audiences, who seemed to "get" what we were doing more often than men. Working on a game for an audience that was a different sex than our all-male team further enriched our learning experience. We couldn't make a game that would be fun for just us, or other similarly-minded men. We had to constantly determine whether the choices we were making sense for a broader audience.

Painterly Rendering. Painterly rendering is a technique where instead of using pixels to render geometry, brushstrokes are used. A brushstroke can be thought of as a billboarded particle, with size, color, and direction. Every piece of geometry then, can be thought of as a particle system. The net effect of this is that a number of different painting styles can be simulated, from watercolor, to impressionism. In a game where the principle mode of interaction is painting, this was ideal.

Out of all of the technical features in *Romeo and Juliet*, the painterly rendering is the biggest jaw dropper. When we first started conceiving the game, one of the things that we quickly agreed on was that we wanted the game to have a distinct look. One of the things that we had learned is that it is easier to "break" new gameplay, if the game has a technological "wow" factor. So, instead of the going the usual photo-realistic route, we decided to look into painterly techniques.

At the start of the project, there were concerns about the viability of painterly rendering. The examples we had found of it on the Internet we either non-realtime, or were using models that were extremely low poly, at low frame rates. No example that we found used painterly rendering with dynamic meshes. To be honest, we weren't sure if we could pull off the rendering at a decent frame rate, if at all.

Ironically enough, the technological feature that we were the most unsure of, was quickly finished in three weeks. This was an incredible morale booster. There is definitely something to be said for having something cool on the screen.

Outside Help. Coming into *Romeo and Juliet*, the team's collective knowledge of ballet amounted to "I saw the nutcracker once when I was seven, I think". Considering that the team was doing a ballet game, this was not a good situation to be in. Luckily, our teacher, Stephen Murray, knew a dancer at the Orlando City School of Ballet, and introduced us. The dancer agreed to come to the school and be videotaped doing moves, and answer any questions that we had about ballet. With the efforts of that dancer, and the book *Ballet for Dummies*, our team was able to learn the basics of ballet in a very short timeframe. The videotape that we filmed also proved to be invaluable as a base for our artist.

Our team also had another large gap in terms of our asset creation, as we had no one to do animation and modeling. The amount of animations that we needed was huge, and they needed to look believable. Luckily, one of the best modelers and animators at Full Sail, if not the best, Rory Young, agreed to help us. Even though he was doing work for four to five other projects at the same time, he was style able to turn out one high quality animation after another. Animations turned from one of the high-risk areas of our game, to one of its shining points. There are very few people that could have done the job that Rory did, and we were very luckily to have him aboard.

Clear Ideas on How to Implement Tech. Being enrolled in the Bachelors completion course at Full Sail, all of the members of the team had worked on a 3D game project before. Even though the previous projects we had all worked on had only lasted three months, there is no substitute for experience. Universally, every team member had had the experience of making wild, to the point of useless, guesses on how to implement tech, and how long it would take. And after that tech had been implemented, having a nightmare getting it integrated.

This time around, we were much better prepared. From the get-go, the team had a clear idea of how each piece of the game would fit together, and we also allocated enough time for each piece to be completed.

A large part of this can be attributed to the longer amount of time we had in pre-production compared to the previous time through. Instead of only one month, we had two. This may not seem like much of a difference, but a single additional month in pre-production can make all of the difference in the world.

As part of the final project course, the second month of pre-production was used as a "proof-of-concept" stage. One of the requirements of this month was that each team prove that their game was viable, in terms of gameplay and technology. In order to fulfill this requirement, our team developed demo applications showing off painterly rendering, smooth skinned model exporting and playback, interactive music, and a text based mock-up of our gameplay. These applications were then shown to the studio and faculty, who then made the final determination on whether or not the project would be allowed to proceed.

Having prototypes of tech this early in the development process helped get a good grasp of how the pieces needed to fit together. For instance, for proof-of-concept, the painterly rendering was populating a static mesh of a torus with brushstrokes. How would we go about rendering a mesh that was being deformed? In hindsight, this is an obvious question. Having a prototyping stage let us develop this hindsight at the start of the project, as opposed to the end.

Lua. Lua is a lightweight, extendable, and easily embeddable scripting language. We chose Lua because half of the team had used it before, and we felt comfortable with it. Lua has a number of features that make it ideal for game development. Lua's main data structure is the all-powerful table, an associative array, which the entire language is centered around. Combined with the fact that functions are values in Lua, this enables you to do some very powerful things, with very little code. For instance, we handled action mapping at as a Lua table of functions, that was indexed into using standard DirectX #defines. This was nothing really that special, except for the fact that the entire implementation totaled about five lines of Lua code. Not bad.

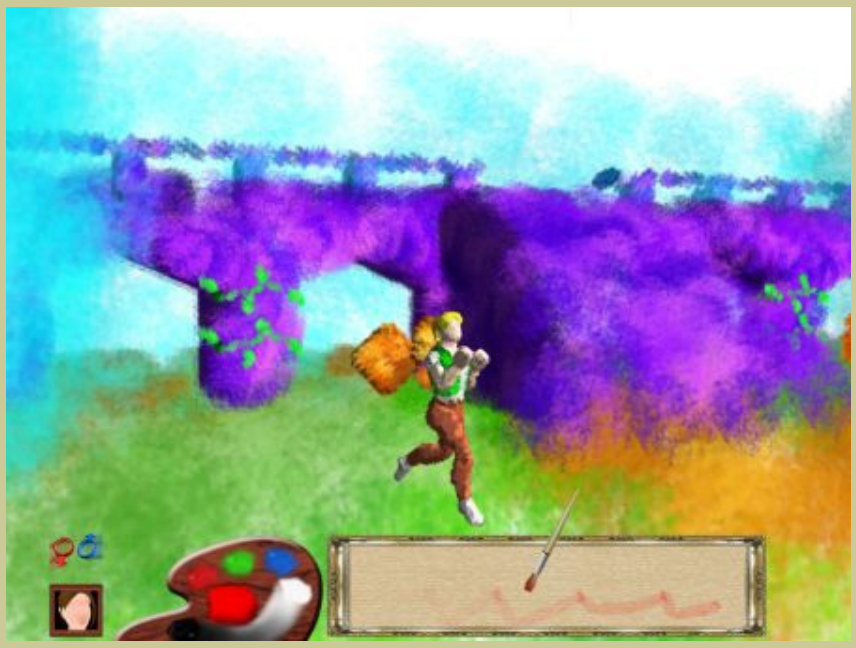
A very large portion of our game was written in Lua. From the main game loop, to menus, to animation state machines, to just about anything that didn't need to be fast. This main advantage of this, was that when we needed to make changes to gameplay (which was very often), we could do so without compiling, and often times, from within that game itself. This made our life, much easier.

What Went Wrong

Lua. Of course, the last thing to go right was the first thing to go wrong. Many of Lua's positive aspects, are also its biggest shortcomings. Lua is a very simple language, and is very quick to big up. One of the reasons for this is that it is a loosely typed language. As a matter of fact, variables don't have type, only the value that they contain does. This has its advantages. For instance, you can do things like assigning the name of a function to a variable, and then executing it by adding parentheses, for instance. It also has major disadvantages, making a typo in a variable name will create a new variable, and on top of that, Lua will not throw an error. This isn't such a big deal with a small codebase, or when the domain of Lua's use in a game is limited, but when a large portion of the game is written in Lua's, it can became a hassle. Often times, a member of the team would make a seemingly simple change, only to be greeted by *Romeo and Juliet* standing still on the stage, doing nothing, with naught a peep from Lua.

Another issue we had with Lua was its reliance on tables. Just about everything in Lua is a table, including the global namespace (which is a table). By default, all variables are added to this table, unless prefixed by the "local" keyword. This didn't seem like a problem at first, but by the end of the project, the global namespace was so polluted that dumps of it were useless for debugging purposes. Reliance on tables hurts Lua in other ways, for instance, in implementing OOP. Yes, in theory, you can have some OOP features in Lua, but it's kind of like the libraries that offer functional programming features for C++. Yes, they work, but they're rather ugly.

All of the problems we had lead to basically to one point. Lua's ease of use and lack of imposed structure make it great as a scripting language, but lacking as a programming language.



Romeo expressing pride

Gameplay. The most overwhelming issue we had with *Romeo and Juliet* was the gameplay. All of the technical goals we had in mind for the game were comparatively easy to accomplish. After the class had approved the high concept for the game, we had problems conceptualizing just what the gameplay be like. Our first meetings were punctuated by blank stares, and a sense of "What have we gotten ourselves into?"

Eventually however, we were able to come up with enough of a design to get a prototype going. The player would express emotions to, and in response to, the other dancer, using the paintbrush. The other dancer would have a psychological model of their personality, which would dictate how the dancer reacted to the player, which would in turn affect the overall relationship.

This seemed like it might work. So we quickly constructed a prototype of it, using the paintbrush interface, interactive music, and text to provide feedback on the emotional and relationship states of the game. The prototype provided the team with the first glimpse of what the gameplay would be like. Painting with the brush was fun, and so was watching the emotional states of the dancers (the moods of the dancers swung so much during the prototype, that the phrase "trailer-park romance" was tossed around quite a bit). Listening to the music change in response to the overall relationship status was also cool. But there was a bigger problem, after the initial few minutes of gameplay, the game quickly became boring. On a few of the critiques we got from our classmates, it was noted that the game was basically randomly scribbling with the mouse, and didn't seem very engaging. This was more or less correct.

With three months left to go, we had to find a way to make the gameplay work. We quickly decided that we needed to give the game some sort of narrative. We came up with the idea of splitting the scene into "sub-scenes". A sub-scene would be an emotional paradox, which the player would have to work through using his emotions. For instance, Juliet would feel deeply in love with Romeo, but also would feel that the relationship was moving too fast. As Romeo, how would the player work his way through this paradox using the paintbrush? We never satisfactorily worked that part out.

Research and lack of time. Five months is an insane amount time to try to be successfully innovative, even if you are well prepared and have done your homework. Trying to research, design, develop, and test a game as different as *Romeo and Juliet* turned out to be impossible to do within that timeframe.

With such a short timeframe, we had very little time to research. Topics that should have been given many months of research, such as emotional agents, were given a few days. This led to us making quick, fly-by-the-seat-of-our-pants decisions that couldn't be easily backed off if they proved to be incorrect. We tended to choose easier solutions that could be implemented in less time, over solutions that would have probably ended up being more correct in the end. Many tough technical hurdles have to be passed for an interactive love story to be realized, and it just wasn't possible to hurdle them in such a short time frame. Regardless, I feel that the project yielded an interesting and educational result in the end, but with more time, I think that we could have run much further with it.

Asset Problems. We knew that one of the biggest risks coming into our project would be getting the amount of assets that we needed. Although our animator was extremely capable, there was only one of him, and the one of him that there was, was working on several projects at once. Because he was spread so thin, this meant that often times, he didn't have the time to do a second pass on animations that needed tweaking/modifications. It also led to us having to cut down on the amount of assets that we needed.

One of the issues we ran into is that it is extremely hard to nail any asset requirements down when you are working on a game that is experimental in nature. The only assets that we knew we needed for certain were ballet animations, beyond that, we didn't have any solid list of assets. This made it hard for the artists to schedule in time to do our assets (At Full Sail, assets for final project are often done by interns and lab instructors). It also to led to us having to ask artists to redo content multiple times, through no fault of their own. Unfortunately, I think that this just comes with the territory, when you are attempting to do something that hasn't been done before.

Player Feedback. To get a game player to react to a game on an emotional level, let alone interact with, you have to have a way of effectively conveying emotion to the player. Emotion is conveyed between persons in real life through voice inflection/tonality, and body language. This is common to almost all forms of popular entertainment, from literature, where your imagination plays the speaking parts for you, to music, where the music is an emotional journey.

Ballet is a very emotional art form, through dance and mime. However, getting that emotion to come through in real-time animation is a very difficult proposition. Doing realistic ballet animation is difficult enough as it is for an artist. There is a huge difference between asking an artist for a "pirouette", versus a "slightly sad pirouette". Only very recently has computer animation gotten good enough to believably convey emotion in cinema (witness the difference between the characters in *Final Fantasy*, and Gollum in *LOTR: The Two Towers*). Getting that same sort of emotion, interactively, and in real-time is at least a few years away.

Our solution to expressing emotion was to have the dancers do mimes. For example, a dancer could mime "Love" by placing his/her hands on her heart. These mimes could be combined with other animations, so that we could have a "sad pirouette", in a manner of speaking. Unfortunately, this didn't really do the trick.

Besides the problems with getting the dancers to convey emotion, we also had problems conveying the current state of the game. It is often

times very hard to understand exactly what is going on in the game. A player's understanding of the current game state is related to how connected the player feels with his/her avatar. Identifying with a character that you only see from the balcony, and that you control solely through painting, is a tough sell.

Conclusion

Although flawed in many ways, we are proud of how *Romeo and Juliet* turned out. Though it's not currently available for purchase or for free distribution online, the game is as different as they come, and although it doesn't deliver on its promise of being an interactive love story, there isn't another game that looks, sounds, or plays like it. For a group of six students with no professional experience, I think that's a pretty big accomplishment.

Game Data

Number of full-time developers: 6
Number of part-time developers: 3, one 3D artist, one 2D artist, and one audio engineer
Length of development: 5 months
Release Date: September 2nd, 2004
Target Platform: PC
Development Hardware: 1.7ghz Pentium IV Mobile laptops with 512MB RAM, ATI Mobility 9600s.
Development Software: Microsoft Visual Studio .NET 2003, Photoshop, Maya, UltraEdit, Pro-Tools, DirectMusic Producer.
Notable Technologies: Painterly renderer, smooth skinned skeletal animation, interactive music.
Project Size: 581 Files, ~25,000 lines of code.

Return to the full version of this article