GAMASUTRA
The Art & Business of Making Games

Postmortem: Magnin & Associates' *Skittleball*

By Ed Magnin,Paul Schorn

[In Gamasutra's first postmortem of an iPad game, two members of the team at Magnin & Associates, a veteran developer of portable games such as Prince Of Persia for Game Boy Color, discuss the difficulties and surprising benefits of developing Skittleball for Apple's new platform.]

We based our design of Skittleball loosely on the British game of Skittles, only using a ball instead of a spinning top to make it less random and more interactive. The player tilts the iPhone or iPad to roll a ball through a series of rooms, avoiding the black holes, while trying to knock all the pins down in order, in the fastest possible time.

Our main design goal was to take advantage of the excellent 3D graphics capability of the iPhone along with the built-in accelerometer to make unique game play for players of all ages.

We decided to offer two very different gameplay modes: in the top-down view you keep the device parallel to the floor and lean it slightly to make the ball roll (similar to Labyrinth). In landscape view, you hold the device parallel to the wall, and lean the top edge away from you to go faster and lean it to the left or right to steer (like a racing game).

During development we decided to let the player switch between them at any time by just tilting the device in the new orientation.

This product had been in development since late October for the iPhone. A couple of weeks before its release Apple sent us an email saying that if we submitted by March 27 at 5 PM, it would be considered as a launch title for the grand opening of the new iPad App Store when the iPad went on sale on April 3. Naturally, with the game more than 90 percent complete, we decided to quickly adapt it for the iPad.

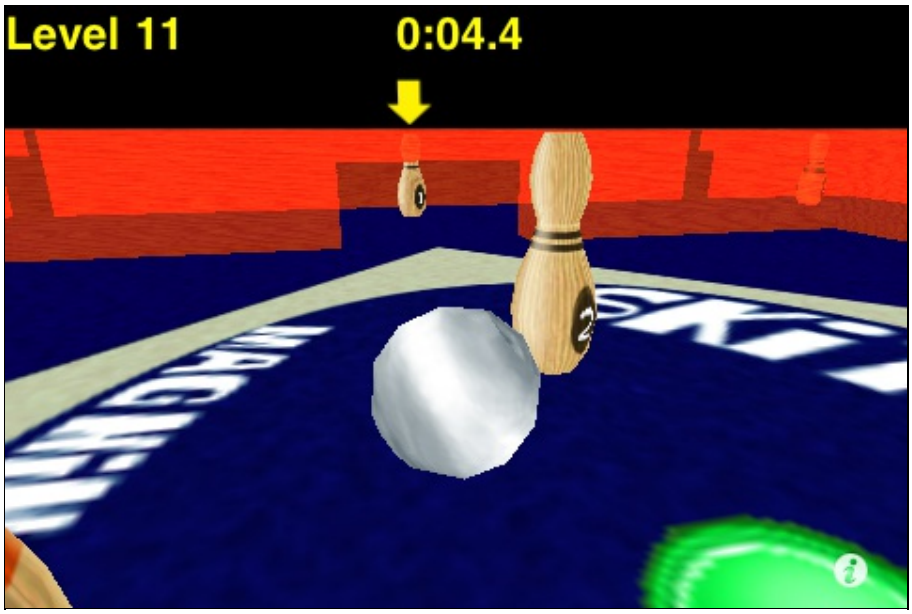
Our dev team has a wide range of experience -- from recent college game programming grads to 30-plus years in the industry. Magnin & Associates has specialized in handheld platforms since its inception in 1993, but had just switched over to iPhone development in the past year. At the start of this project we had a couple of previous games in the App Store, with an additional free version.

We decided to write our own game engine this time, although we had reviewed a number of excellent candidates. It was our feeling that we would learn more, have more control over the source code, and hopefully have a competitive edge as we move forward with additional projects. (A second game was started slightly after this one, also using the same engine.)

We used OpenGL ES for our 3D rendering and OpenAL for audio. We relied heavily on Apple's sample code for examples on how to communicate with the hardware. During development we learned some of the nuances of Objective C.

While there are now lots of books on how to develop an app for the iPhone, and even a few on developing games, very few if any mention 3D or OpenGL ES. And the books on OpenGL ES are not necessarily specific to the iPhone.

As a game developer with two previously-developed 3D games on the Nintendo DS, we wanted to do the kind of quality 3D project we thought we were capable of.



What Went Right

Very often in game development, the things you think are going to be easy turn out to be much harder than expected, and the things you thought were going to be hard turn out to be easier than you thought. The following problems turned out to pleasantly surprise us:

1. Sound

We needed to implement sound in a way where we could get the best quality with using as little space as possible. To accomplish this we used wav files for the sound effects and MP3s for the songs.

We decided to mix some of Apple's base sound framework with OpenAL to get the best out of both the effects and music. We were able to have our sound manager up and running in under a week. This included the ability to modify volume and pitch on the fly. We were all surprised with how well it worked and how fast it was to implement.

2. iPad and the simulator

Programming for a new product is hard, but programming on a product you don't have yet -- precluding real-world testing -- can prove to be even more of a challenge. We wrote most of the game with the intent of making it an iPhone/iPod Touch game, but when we got near the end of production and saw that we had a chance to get in on the opening of the iPad store, we made a push to make the changes needed to make it an iPad game.

When it came down to testing the changes, all we could do to see the changes was to use the iPad simulator to try and test as best we could. While we could adjust for the new larger screen size, we could not actually play the game since the simulator does not simulate the accelerometer.

In the end, it turns out the simulator was enough to help us get all the changes we needed in on time. When we went to pick up our first iPad on release day, we literally couldn't wait to get back from the Apple store to see how it played on the iPad. We actually purchased a copy of our own app online at the App Store, to make sure everything came out the way we expected it to.

---

3. Graphics pipeline

Having an experienced art director who is fluent in 3dsMax, we wanted to avoid the tendency of most iPhone startups to switch to some less expensive alternative. We found a Collada2Max exporter that gave us an XML .dae file. We then wrote our own preprocessing program as a native Mac application to convert the model's verts, normals, and texture coordinates to const arrays and a model3d struct.

We were pleasantly surprised how easy it was to write a native Mac program in C using XCode. Installing a new model in the game is as simple as including (#import) the header file and then referring to the model by its name.

4. Level design

After getting the proof of concept with a completely playable level of the game up and running we made a simple level creator function. After making a few sample levels on our own, we let our QA Lead, who is studying to become a game developer, lay out most of the thirty levels in the game.

Once she gave us a file, we would build the game and send it back to her to test. Neither she nor our art director were using Macs so we had

to provision their devices so they could drag the latest build onto iTunes and then sync to see it running.

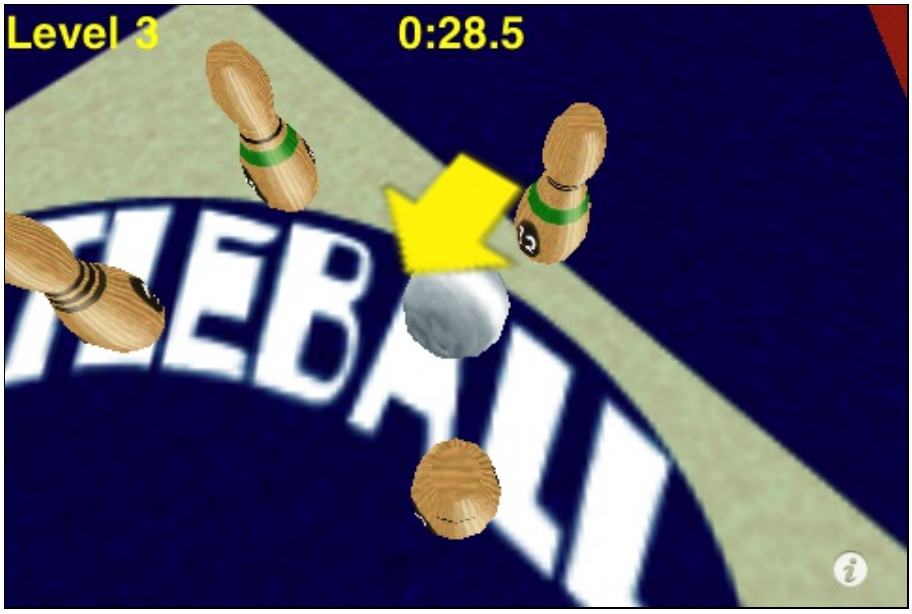
The quick feedback they get by being able to run it on their own iPhone contributed to the quick turn around we got on fixing any problems they noticed.

5. The final countdown

At the end of any project in the game industry you expect there to be a bit of a crunch time. We entered crunch time with more problems that you would want and more unfinished game code than you would expect. Part of this is because shooting for the iPad release was a last-minute decision.

But even with all that standing in our way, we still managed to post the build onto Apple's website just in time to make the launch of the iPad store. Most of it just boiled down to hard work and long hours, but in the end we would not have made it in time if it weren't for two things: luck and fresh eyes.

Right as we were going into crunch mode, we brought in another experienced programmer and he seemed to catch everything that went through the cracks and gave us that little boost to get over the last bit of the hump. It's important to have an extra set of trained eyes to question and challenge how and why things were coded a certain way. It makes everyone really think about the best way to get something done.



What Went Wrong

1. Ball rolling

You would think that having a ball roll on a surface would be a trivial problem. It was fine along all four walls of the room, but once we started testing it diagonally it started wobbling annoyingly.

After a lot of Googling, checking reference books, and experimenting with various calls using OpenGL's glRotatef function, we realized we needed to rotate the ball in both the X and Y axes at the same time. (Z points up in our world.)

We had to load an identity matrix, rotate it on the X and Y axes the amount we needed to roll diagonally for that frame, and then save the matrix. Then we had to multiply the current cumulative ball rotation matrix times the newly calculated recent move matrix.

2. Graphic corruption problem

We started noticing the ball corrupted, as if the triangles that made up the sphere had come unattached on one or two points. We accidentally discovered that it corrupted only the first model in the draw list, so we took the expedient way out and left an unused model at the beginning of the list -- not our proudest moment, but something you do to make it work until you can figure out what is actually causing the problem.

As the code got more complicated, it took a larger and multiple models to keep it from corrupting the ball. We finally put our rendering code under a microscope, caught some coding problems, and switched to using const arrays in our graphics header files.

Having worked on cartridge games, we were used to having large amounts of graphics data that remains in the ROM where a pointer simply references it.

Our new programmer questioned whether the iPhone was trying to move large amounts of data onto the stack. As a const it will stay in one

place.

---

3. Getting 2D and 3D to coexist

Our game is built in a 3D world, but some of our biggest problems came from 2D. More to the point, we had trouble writing in different layers and getting out GUI to work properly.

We studied all the example programs we could find on Apple's forums or elsewhere. We found great sample code for separate 3D apps and 2D apps and even some GUI examples, but we had a really hard time finding a simple example of the two working together.

After going though many websites and some video tutorials we found a way to patch the two sections of code together. Who would have thought a simple problem like putting a 2D sprite, or timer, on the screen in front of 3D OpenGL models would be such a problem?

4. Running into a wall

There is always going to be times when a team runs into those problems where they just can't seem to find the right solution. Running into the wall is always a problem, but we allowed it to stall the momentum of work on more than one occasion. We would come together as a group and pound on the problem from every possible angle until we cracked the problem.

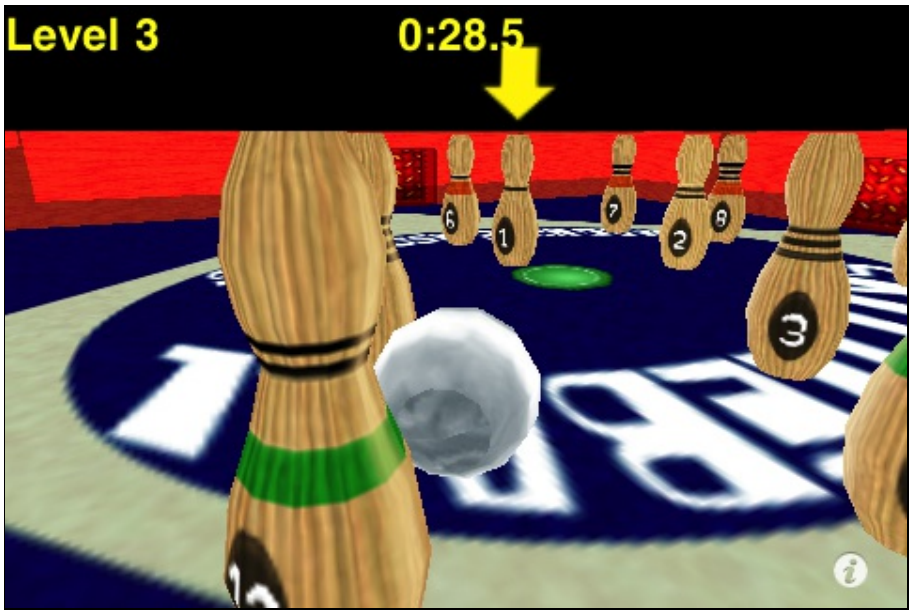
While this did end up solving the problem over time, it also threw us off our rhythm and kept us from working on other areas of the project where we could have actually made some progress. In the future, we should probably have one or two programmers focus on the problem and allow the others to continue to make progress in other areas.

5. Racing the clock

Football is played for an hour on a 100-yard field, but near the end becomes a game of seconds and inches. That last weekend we were constantly aware of our 5 PM deadline. Having successfully submitted three prior apps, we were well familiar with the process and had already posted the game description, screenshots, and product details.

What we were not familiar with was how creating a "universal" App would complicate it. A "universal" App runs on both the iPhone and the iPad but automatically adapts for the current device. We kept uploading our binary and waiting for Apple's website to evaluate it. Then after waiting several minutes it would complain about missing icons. Apple requires different size icons to post the app.

Unfortunately the information about how to do so was a little sketchy, and difficult to find all in one place. We eventually discovered Apple's Application Loader tool which evaluates the binary locally first and lets you know if there are any problems, before the several minutes it takes to upload.



Parting Advice...

Document everything you do! We keep a progress.rtf file right inside the project so it is there when we are editing the code. We keep a list of the things we need to do at the top and then each thing we do to the game in reverse chronological order. Whenever we have a problem we can go back and see what we did the last time we had a similar problem.

Also, whenever you find you have a build so unstable that you're better off going back to an earlier backed-up version, you can simply refer to the more recent progress.rtf file to see what things you still need to do to that build.

You can then do them carefully one-at-a-time to make sure you don't induce the same problem. We also use a DiffMerge program on the Mac to compare entire project directories between a current and previous build to find things that have changed, such as accidentally deleting an important line during editing.

Return to the full version of this article