J. Matthew Griffis
TD Seminar: Game Studies
11/30/13

Platform Studies: *GameSalad*

Preface

This paper is in the spirit of the *Platform Studies* book series[1], though it is not an official member. (Should The MIT Press issue a cease-and-desist letter, I will change the title; in the meantime, I carry on.) Specifically this essay is inspired by Montfort and Bogost's Racing the Beam, an examination of the relationship between the technical nature of the Atari 2600 game console and the games produced for it. This look at how the functionality of a platform influences the design of software is core to the nascent discipline of platform studies. In this paper I apply the methodology to *GameSalad*[2], a piece of software designed to help non-programmers create digital games (video games).

A note to the reader: I write about how the design of *GameSalad* itself affects the process of designing with it, and what agenda it may advance. In the interest of full disclosure I must admit that I first approached *GameSalad* with an agenda of my own (though it had nothing

[1] http://platformstudies.com/
[2] http://gamesalad.com/

to do with this paper). Last summer I was asked to work on PETLab's "Activate!" project.[3] Activate! is a curriculum to teach children how to create video games,[4] and at the time consisted of two parts: a) a series of games of escalating complexity and b) tutorials for how to create them. The games and tutorials were created using another piece of software called *GameMaker*; my task was to recreate the games and tutorials in *GameSalad*. I had used *GameMaker* before, but never *GameSalad*, so the re-creation of the Activate! games became a way to learn how to use *GameSalad*.

Anyone who has ever attempted to port software from its original platform to another knows the challenge and frustration inherent in the enterprise. It's the same as translating between spoken languages--things don't match up exactly, and making it work requires a lot of rejiggering[5]. I didn't approach *GameSalad* with the mentality of "Let's see what this software can do!" but rather "Let's see how easily this software can do what *GameMaker* does!" Furthermore, I had just spent a year learning how to program, which is the kind of knowledge *GameSalad* exists to circumvent. I was no longer the person for whom *GameSalad* was intended.

Given these factors, could I help but be preemptively biased against *GameSalad*, keen to identify its weaknesses? Perhaps not. However I also think those factors were a perfect primer for writing this paper, because they got me thinking very analytically and critically about how I perceive *GameSalad* to work and how that influences game development using the software.

I don't claim to have mastered *GameSalad* by any stretch of the imagination. There are some features I have ignored, for reasons I will discuss later, and it's possible that their use could address some of my points. I've used the software for about half a year, and am still learning its capabilities.

[3] "PETLab" is The Prototyping, Education and Technology Lab (http://petlab.parsons.edu/).
[4] http://petlab.parsons.edu/project/activate/
[5] A technical term.

Furthermore, I don't mean to imply with this wind-up that *GameSalad* does not have much good to offer. *GameSalad* has many strengths, and I intend to discuss those. *GameSalad* markets itself as a professional tool and charges a premium to use it as such. Consequently I evaluate it both as a learning tool for something like Activate! and as the serious development tool it claims to be. What humble insights I have, I offer here.

A Technical Note

I worked with the Mac version of *GameSalad*[6] (version 0.10.4.1 Beta), except for a brief dalliance with the PC version[7] (also version 0.10.4.1 Beta). As just indicated, *GameSalad* is still in beta. It may change substantially before formal release. Unless otherwise noted, all images are from the Mac version described here.

I. Historical Introduction--Rise of the Middlemen

It is not easy to create games, and it is even more difficult to create video games, requiring as they do a high degree of computer know-how. One must be able not only to imagine and plan out the contents of the game, but actually to program it, and to fix it when it inevitably breaks. For this reason video game development has long been the realm of only those individuals with the necessary computer science background. The average Joe or Jill with an idea for a game simply didn't have access.

That changed with the advent of middleman software. These tools (which are not restricted to the realm of video games) hid the overwhelming arcana of pure code inside a friendly and approachable graphical user interface (GUI) with drag-and-drop functionality, similar to the kind people used every day on their personal computers. Suddenly it was no longer necessary to know how to program well, if at all! Instead, users could simply drag and drop visual representations of objects and logical relationships into place, and the software would invisibly convert that into the actual code.

[6] http://gamesalad.com/download
[7] http://gamesalad.com/download?platform=windows

In 2000, a version of the *RPG Maker* series[8] came out for the PlayStation. The software was like a fully-stocked kitchen, including all the ingredients and recipes to make a 2D role-playing game (RPG). Aspiring designers could assemble their own maps (i.e. levels/environments) from the included tiles, set parameters for how the combat system would work, and use a simplified scripting language to define behaviors and create quests for the player, then save the finished product to a memory card and give it to others to play.

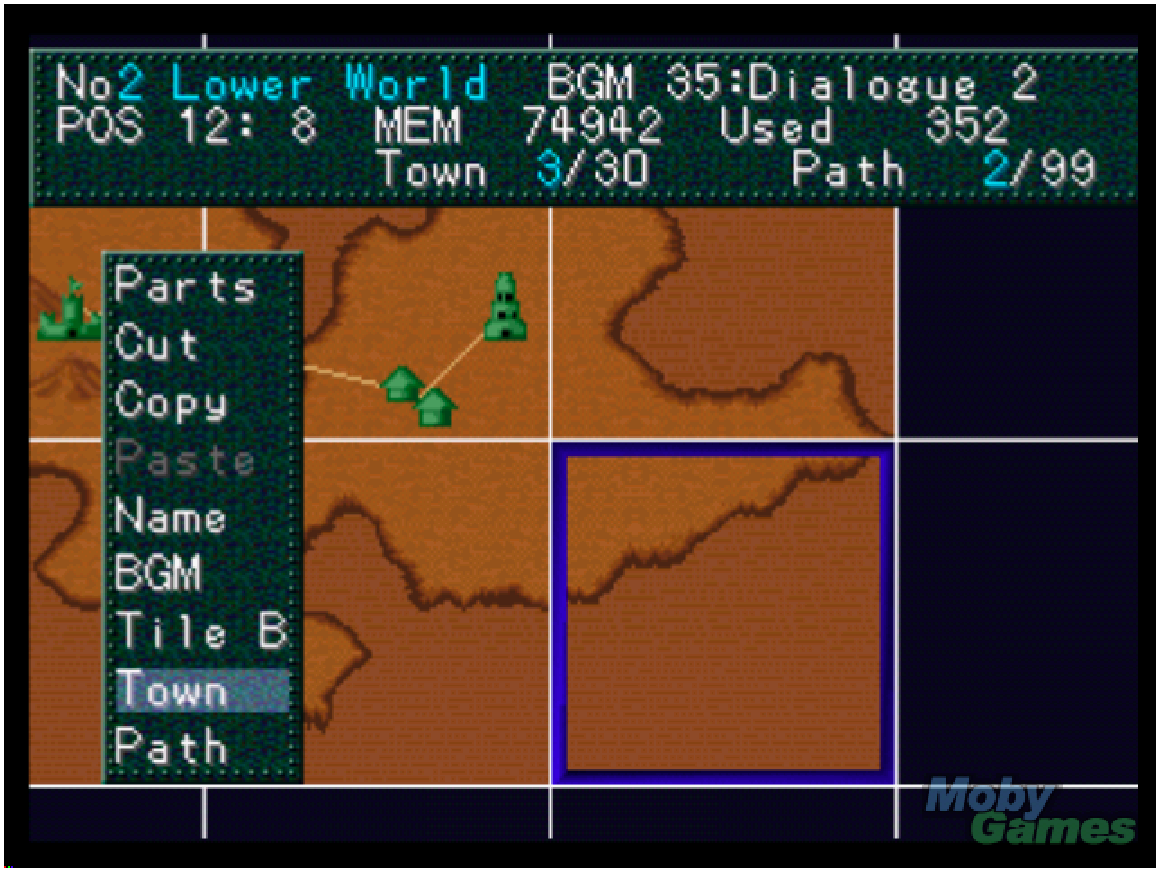


Fig. 1: Assembling a map in the PlayStation version of RPG Maker.[9]

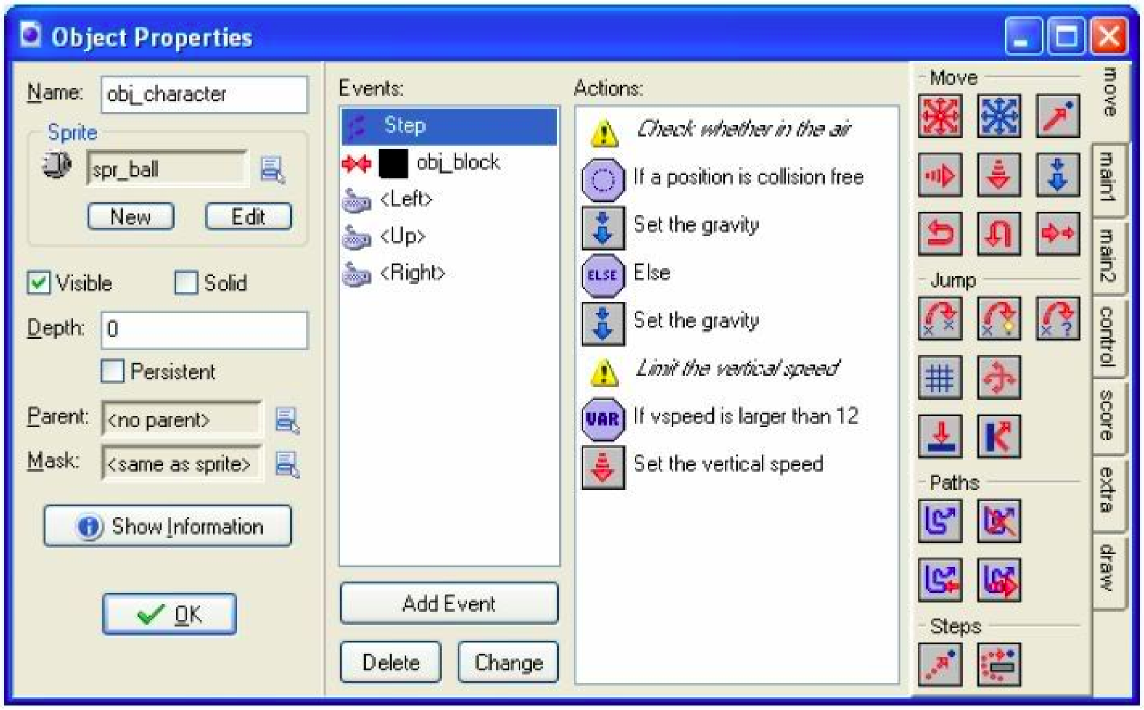
RPG Maker turned consumers into creators. When it appeared on PlayStation the entire user base was likely to be interested in this conversion, since they were all gamers and it's a rare gamer who doesn't dream of creating her own game. The familiar interface of the game

[8] http://www.rpgmakerweb.com/

[9] http://www.mobygames.com/images/shots/l/177222-rpg-maker-playstation-screenshot-overworld-editors.png

controller may also have contributed to accessibility. On the other hand, not all gamers like RPGs, and the software was "RPG" Maker, not "Game" Maker.

One year earlier, in 1999, *GameMaker*[10] came out for the PC. Since then it's received many updates and a huge increase in popularity. Like *RPG Maker*, *GameMaker* allows one to create games using a drag-and-drop GUI--no programming required.



Name: obj_character
Sprite: spr_ball
[✓ Visible] Solid
Depth: 0
Persistent
Parent: <no parent>
Mask: <same as sprite>

Events:
Step
obj_block
<Left>
<Up>
<Right>

Actions:
Check whether in the air
If a position is collision free
Set the gravity
Else
Set the gravity
Limit the vertical speed
If vspeed is larger than 12
Set the vertical speed

Fig. 2: Setting an object's behavior in GameMaker *by drag-and-dropping action icons.*[11]

Unlike *RPG Maker*, *GameMaker* doesn't limit the input and output to the tropes of one genre. Furthermore, anyone can download and use a version of the software for free, though it is necessary to buy various licenses to get access to advanced features and publish games for profit. *GameMaker* is robust and has become a respected tool for creating games, even at the professional level. For example, the wildly successful independent release *Hotline Miami*[12] was created using the software.[13]

[10] https://www.yoyogames.com/studio
[11] http://img.photobucket.com/albums/v369/David_Manning/Operation_Block/GameMaker_DD.jpg
[12] http://hotlinemiami.com/
[13] http://www.yoyogames.com/news/131

II. The Goal of this Paper

In recent years there has been an explosion of middleman software for game development, targeting all sorts of different ends, from high-end 3D environments via Unity3D[14]; to sophisticated 2D games via *GameMaker*; to interactive narrative via *Twine*[15]; to a focus on accessibility for children via *Scratch*[16]. *GameSalad* is another such tool, which offers functionality similar to *GameMaker* but with a greater focus on accessibility and an aggressive push toward ease of distribution across all platforms, including online via HTML5.

Generally speaking, tools like these are available to everyone with a computer and an Internet connection, cost little or nothing (at least initially), and require relatively little education to start using. They democratize game development. For the first time, it is possible for virtually anyone with an idea for a video game to actually create it. We live in exciting times for the video game form.

However, these software are not without flaws. There is a tension between accessibility and control, and ease of use requires compromise. *RPG Maker* only comes with so many graphical assets, behaviors and parameters. It is not possible to create *any* RPG one can dream up because there are limitations on what the software allows. *GameMaker* offers its own programming language--"GameMaker Language" (GML)--for advanced users who want to go beyond the limits of the GUI, but it too has limitations. So does every other such tool, including *GameSalad*.

Note that this is inevitable. I am not stating it as grounds for condemnation. If I write a library of complex code so that someone else can run it using simple code, he or she is stuck with whatever methodology I chose. This is not different than if I paid someone to mow my lawn and didn't like the lawnmower handling--if I wanted it done "just right," I'd have to mow my own lawn. The only way to assert complete control over every aspect of game development

[14] http://unity3d.com/
[15] http://twinery.org/
[16] http://scratch.mit.edu/

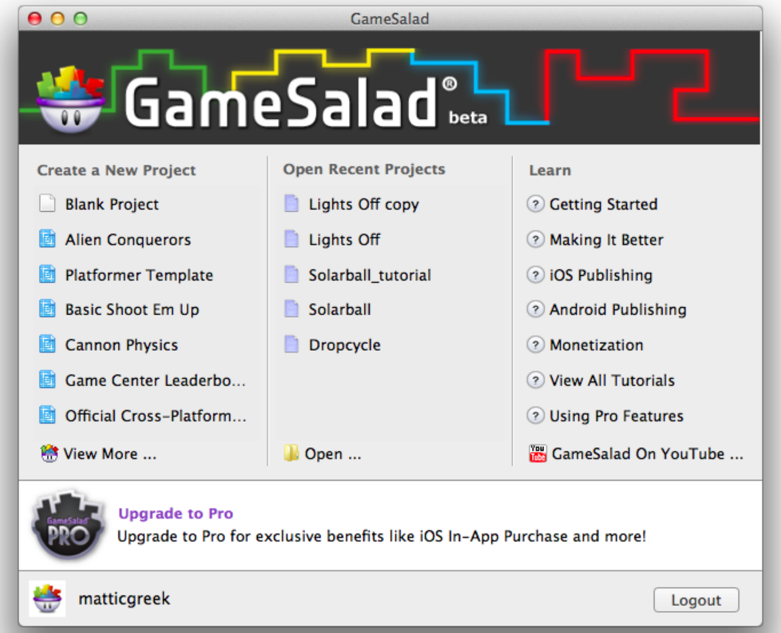
is to learn to program and write the game from scratch. Otherwise, one takes advantage of other people's tools and accepts the loss of some control.

But! It is important to be *aware*. The technical limitations quickly become apparent when one tries to accomplish a certain thing and discovers the system doesn't support it. This is frustrating, though (again) inevitable. However, any piece of middleman software also has an *agenda*, conscious or not, meaning that its creator had ideas about how things should be done, and developed the tool to reflect those ideas. Using a tool like *Unity* or *Scratch* or *GameMaker* means operating within a theory about what a game is and how it works, and it is not always easy or even possible to step outside that theory.

The goal of this paper is to look hard at one specific piece of software, *GameSalad*, and try to figure out its technical limitations and its implicit agenda, and how these affect the games produced using it.

III. Getting Started with GameSalad

Let's begin at the beginning. We've downloaded *GameSalad*. Let's open it!



Upgrade to Pro for exclusive benefits like iOS In-App Purchase and more!

Fig. 3: Upon opening GameSalad.

Well, this seems quite friendly! On the left we can create a new project, from scratch or from a series of included templates that sound like they demonstrate familiar genres and specific functionality. In the middle we can open existing projects,[17] and on the right we can learn how to use the software. There's also a "Login" button (or in this case, "Logout" since I'm already logged in)--GameSalad has a strong focus on community, and people are encouraged to create an account, not only to publish games on the GameSalad Arcade but also to participate on the forums. I'm also encouraged to upgrade to the Pro version of the software, which (surprise!) is not free.[18]

Regarding the Learn section, it's telling that after two entries, "Getting Started" and "Making It Better," the suggestions jump immediately to publishing for mobile and monetization. The monetization is via "In-App Purchase" (IAP), which requires iOS...and a paid Pro account. There are actually quite a lot of tutorials covering much useful information. One can see them all by clicking "View All Tutorials" near the bottom, and discover that "Getting Started" and "Making It Better" are really categories that include all the tutorials related to the game development itself (pre-distribution and monetization). But it is clear from the get-go that this is a tool for making money from the lucrative mobile games market. Fair enough, *GameSalad*; at least you're honest about your commercial agenda.

"But, I just want to make games, and teach children how to make games, and publish games for free, maybe even on the Internet! Can I do that?" Why yes, yes you can! (Though that seems to be deemphasized in the tutorial listings.) I give *GameSalad* a hard time, but the mobile market *is* hot right now, and one can't begrudge a desire to get in on that and help others to get in on that. The question is whether this push toward a specific goal prevents one from pursuing other goals, and the answer is that it does not.

However, the focus on in-app purchases bothers me. *That* is an ideological agenda. It promotes the "free-to-play" model currently in vogue, in which games are free to download and

[17] Pictured: some Activate! games (or at least the names).

[18] At the time of this writing, the Pro version of *GameSalad* costs $299 per year. It is a service.

play, but certain content inside the game is locked behind a paywall. It might be a new mission that the player buys once and then can play as often as desired, or it might be a consumable item which, when used during play, disappears and must be purchased again. Arguing the merits and ethics of this approach is a whole other paper, and indeed many people have written about the topic, but suffice to say that it is only *one* model of monetization. Yet it appears to be the only one supported (or at least promoted) by *GameSalad*.

I suppose there's nothing stopping one from creating a game with zero IAP and then selling it on the app store for money; this is the "Ye Olde Schoole" of monetization, back before there was a word for that.[19] Given market trends, it may be foolhardy to pursue this strategy, but it is a strategy. However, the aggressive promotion of IAP in *GameSalad* as *the* monetization strategy, even from the opening screen (we haven't even made it to the actual UI yet), is a bludgeon of sorts, suggesting that this is the way things are done.

This is not just a question of commerce. It has a great impact on design. In-app purchases are not something to season the game with after it's done cooking, at least not if one wants people to buy them.[20] Rather, IAP must be planned from the beginning and deeply integrated into the game design, especially in the case of consumable items. If they are not required to complete the game, they must convey some desirable benefit, but not one that would unbalance the game in favor of people who paid up, especially in a competitive context (otherwise the game becomes "pay-to-win"). If the purchases *are* required to complete the game, they must be made available through means other than payment of real money--even if those means are exceedingly tedious, such as an in-game currency that the player accrues through play--or it must be clear from the beginning that payment is required to see the game through.

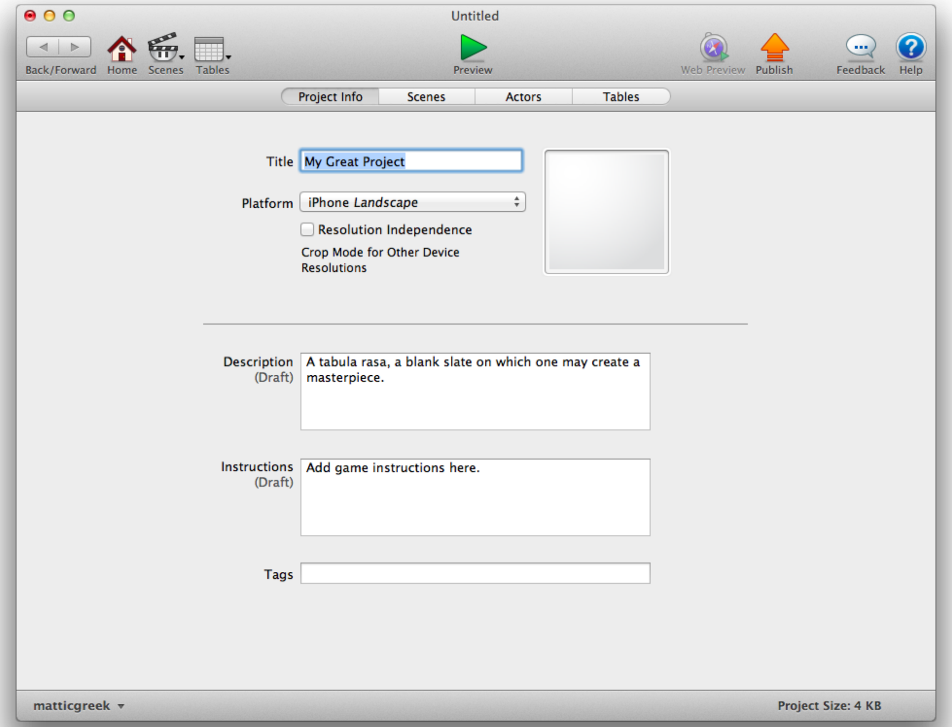
Earlier this year, the website *Kotaku*[21] ran an article speculating what classic game *Super Mario Bros. 3* would be like if it were designed in today's world of free-to-play and

[19] The observant reader will probably guess where I come down on this contentious issue.
[20] Or is Valve. See: *Team Fortress 2* hats (http://wiki.teamfortress.com/wiki/Hats).
[21] http://kotaku.com/

microtransactions. The result is a very different game.[22] In-app purchases are a legitimate monetization approach, but to pretend that they don't influence the design of a game is ridiculous. Anyone interested in releasing games with a different economic model using *GameSalad* has to ignore or actively buck the constant signalling the software sends.

IV. The UI, Part One: Platforms and the Arcade

It's clear the opening screen of *GameSalad* is extremely approachable but has an angle. That being established, let's take a look at the user interface (UI) by opening a blank project.



Title: My Great Project

Platform: iPhone Landscape

Resolution Independence

Crop Mode for Other Device Resolutions

Description (Draft): A tabula rasa, a blank slate on which one may create a masterpiece.

Instructions (Draft): Add game instructions here.

Tags

matticgreek

Project Size: 4 KB

Fig. 4: A blank project.

One thing that *GameSalad* does very well is to be nonthreatening. This UI is very simple aesthetically, with a minimum of buttons on-screen simultaneously and a high degree of

[22] http://kotaku.com/the-horror-if-super-mario-bros-3-was-made-in-2013-for-1168392829

intuitiveness. We may not know exactly what every button does right off the bat, but we can take a guess, and things like “Web Preview” and “Publish” are self-explanatory. We are ignorant, yes, but not overwhelmed. *GameSalad* clearly wants to avoid intimidation, and it cleverly segregates functionality to provide the minimum necessary at any given time. Everything is so big, friendly and clearly labeled that whatever confusion remains feels more like curiosity.

Looking over the contents of the “Project Info” tab, “Title,” “Description” and “Instructions” are clear enough--and “Tags” we feel confident ignoring--but what about “Platform”? Clicking on the drop-down reveals this:

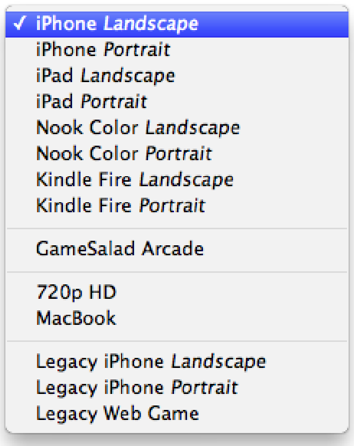
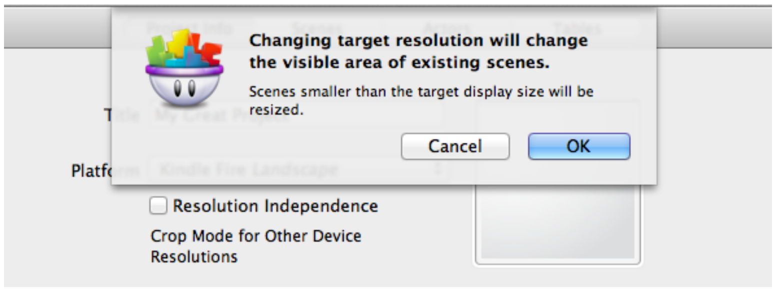


Fig. 5: The “Platform” menu.

“Platform” then must refer to the device on which we intend people to play our game. What are the practical consequences of choosing one? Let’s pick something different:


Changing target resolution will change the visible area of existing scenes.

Scenes smaller than the target display size will be resized.

Fig. 6: A warning about changing platforms.

Ah, it seems that choosing the Platform affects the size of the game screen. It makes sense that this choice comes first; we wouldn’t want to design for the wrong screen size.

Like the highlighted tutorials on the first screen, the Platform menu seems innocent enough but nevertheless takes a certain philosophical stance. On the one hand, it does something genuinely helpful, which is to handle the rote dimensional numbers for us. Can't remember the size of an iPad screen? Don't bother going to Google, *GameSalad* has your back. And it is very important to consider the target platform in game development and crucial to do so early.

On the other hand, look again at that list. Six devices in various orientations (newer iPhone, older iPhone, iPad, Nook Color, Kindle Fire, Macbook[23]), two options pertaining to web publishing ("GameSalad Arcade" and "Legacy Web Game") and one regular-old resolution ("720p HD"). That is an awfully limited list, focused primarily on Apple products and mobile devices. It seems very much in keeping with the previous emphasis on mobile.

One can check "Resolution Independence" and edit settings manually to craft a game at a different size than any of the prepackaged options. But this is extra work. The default setting for Platform is "iPhone Landscape," *not* "--Select a Platform--", as though to help us skip the tedious business of deciding; there is already an implicit statement about the standard mode of operation. Doing things differently means fighting a system that says "these are the platforms for which you should be making games."

A list like that can easily be updated, of course.[24] The options are just shortcuts for specific settings. But the lack of a "Custom" option is a problematic omission. Why is the only shortcut not tied to a platform 720p? Why not make it easy to create a game with a unique resolution? A game like Jason Rohrer's *Passage*[25]--which runs at an extremely unorthodox resolution of 600x96 pixels--demonstrates that there is a lot of possibility in experimenting with aspect ratio. But *GameSalad* doesn't do much to promote that kind of experimentation. If anything, it makes it more difficult.

Some restrictions are especially chafing--for instance, the set aspect ratio of GameSalad

[23] Whatever that means. "Macbook" is a broad term.
[24] And will need to be, because its relevance depends on what's hot in the device world.
[25] http://hcsoftware.sourceforge.net/passage/

Arcade. One of *GameSalad*'s biggest points in its favor is the ease of publishing to the web using HTML5, courtesy of the "GameSalad Arcade" website.[26] Previously getting a game playable in a web browser required some kind of software installation like Flash, or a "web player" that took time to download and might not be compatible with a specific browser, or some other stumbling block. Publishing a game using HTML5 removes all of those concerns, for the fastest, most seamless, universally compatible experience. Huzzah!

GameSalad makes it extremely easy to do this, but one must choose "GameSalad Arcade" as the game's platform. GameSalad Arcade has a resolution of 480x320 pixels. That is the same size as this image:

Fig. 7: It's so beautiful! If only it were bigger than a GameSalad Arcade screen![27]

If we set our platform to GameSalad Arcade, we can publish our game for free to the GameSalad Arcade website (yes!) and easily get the HTML code to embed our game in any other webpage (double yes!). If we make our game at any resolution other than 480x320, we don't get to do these things.

[26] http://arcade.gamesalad.com/

[27] http://www.blackberrygood.com/uploads/allimg/111103/2-1111031203110-L.jpg

Consider that a *small* sprite[28] might be 32x32, meaning we could fit 10 of them stacked vertically in a GameSalad Arcade project. 10 is not many. And if we wanted to use bigger graphics, like the kind commonly found in a computer game, they would take up most of the screen by themselves. 480 x 320 is *tiny*, but that's the only way currently to take advantage of *GameSalad*'s HTML5 publishing power.

As one might imagine, this has serious design consequences. It is not possible to switch the orientation to make an Arcade game taller than it is wide, which makes vertically-oriented games almost impossible to do, unless one uses a scrolling camera. If one does use a scrolling camera, the play experience becomes very different than if the player could see the entire field simultaneously. With the use of a scrolling camera, one can make the game field as large and sprawling as desired, but still has to use tiny sprites, or else design for a very, *very* limited range of vision around the "giant," screen-hogging graphics. Speaking from personal experience, in order to port several of the Activate! games I had to completely redesign the levels in order to make them work on the GameSalad Arcade platform. The result is a similar game but not the same game.

GameSalad Arcade is great, but it is extremely restrictive. This is a shame because the ease of web distribution means the greatest opportunity to share work, yet the work is severely limited by the platform. There is no need for this, at least outside of GameSalad Arcade's own website, and a greater flexibility for web publishing might be *GameSalad*'s most-needed feature.

V. The UI, Part Two: The Scene View

Diving back into the UI, we finally click on the next tab over from "Project Info," which is "Scenes."

[28] "Sprite" in this context refers to a 2D graphic, such as a tree or a person.

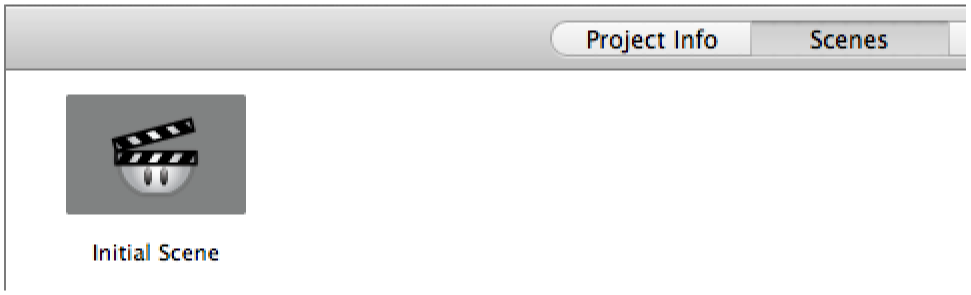
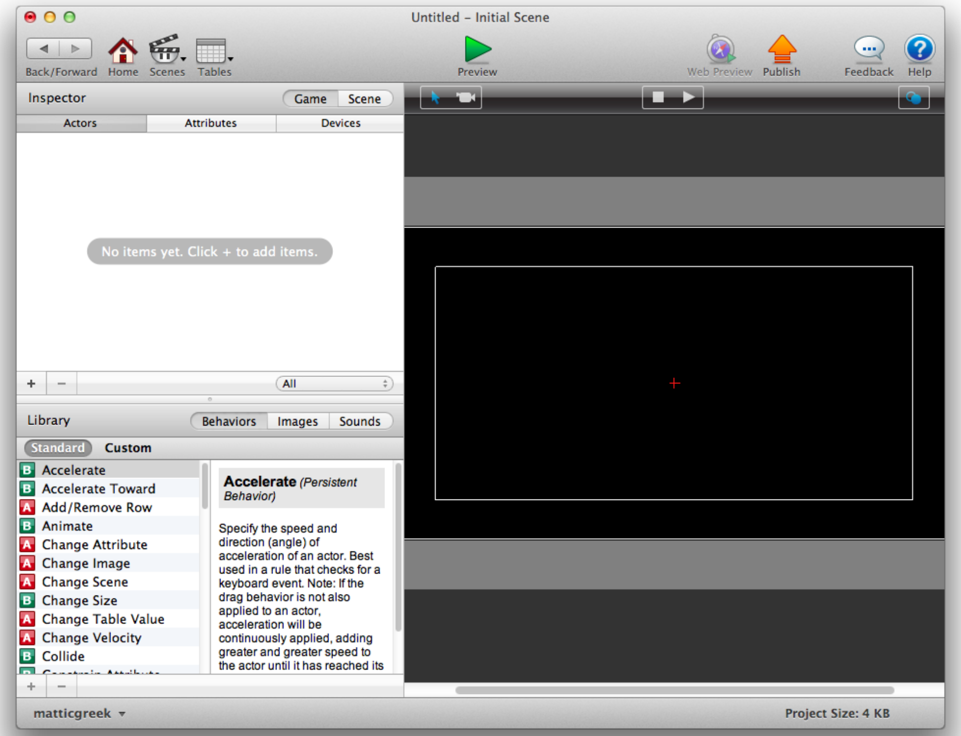


Fig. 8: Clicked the "Scenes" tab.

We're still not totally sure what a "scene" is, but apparently there's an Initial one, so we double-click on it and see the following:



Accelerate *(Persistent Behavior)*

Specify the speed and direction (angle) of acceleration of an actor. Best used in a rule that checks for a keyboard event. Note: If the drag behavior is not also applied to an actor, acceleration will be continuously applied, adding greater and greater speed to the actor until it has reached its

Fig. 9: The "Scene" view.

This has more information than the previous screens, but it's still not overwhelming.

So far I've been writing this description as though I were new to the software and attempting to evaluate how easy it is to intuit, but it's hard to predict how an actual new user would approach learning *GameSalad*--whether she would avail herself of the helpful tutorials, for example--so for the sake of time I will admit that I have knowledge. The image above is of the “Scene” view. The black box on the right is the game screen itself, sized according to the platform selected previously. In the upper-left is the “Inspector,” for creating and manipulating the components of the game, while in the lower-left is the “Library,” for storing the logical, graphical, and audio assets.

Once again, this is all very well organized and easy to understand. Clicking on the “Images” or “Sounds” tab in the Library reveals a blank white box, into which one can drag and drop image or sound files for *GameSalad* to import and make available for use, which it does seamlessly. Each tab also has a link to purchase pre-made content from an online marketplace, content which may have been created by other *GameSalad* users; this is the first glimpse of the other side of monetizing *GameSalad* (and general artistic) expertise, which is making content for other users to purchase for use in their own games.[29]

Up in the Inspector, we see tabs for “Actors,” “Attributes” and “Devices.” “Attributes” turns out to be *GameSalad*’s name for variables, meaning containers that store a changeable value, just like “x” and “y” in algebra. “Devices” grants easy access to the data from the computing hardware, such as the mouse location or the device’s accelerometer. This last is a real boon, especially for mobile devices with touch screens and gyroscopes, making it easy to design gameplay based around those features.

The nature of “Actors” is not immediately apparent (the description refers to “items”) but it quickly becomes clear that the term means the objects that make up the game. Actors must be created in the tab, then may be dragged and dropped directly into the black box of the Scene on the right.

Note that *GameSalad* considers everything put into a game to be an actor, whether it “acts” in a conventional sense or not, from static images to the player’s avatar. This is a subtle point but an interesting one. For example, you can easily create an actor with an attached image by dragging the image from the Library and dropping it into the Actors tab of the Inspector.

[29] The software Unity offers a similar user-driven marketplace.

However, if you bypass the Inspector and drag and drop an image from the Library directly into the scene, the software creates an actor with that image attached anyway. While this might seem strange for a static image, the advantage is that, by making a distinction between an image and the container for that image (the actor), and then letting the designer manipulate the *container*, it becomes possible to control how the image functions in relation to the manipulated container, e.g. if it is repeated, distorted, etc. The image is not a thing in itself but rather a pliable mask, which makes something like tiling (repeating an image) easy to do.

VI. The UI, Part Three: Hello World

A rite of passage for beginning programmers is to draw a rectangle and make it move across the screen, so let's see how easy it is to do that. First, we create an actor by clicking the plus button in the Actors tab of the Inspector.

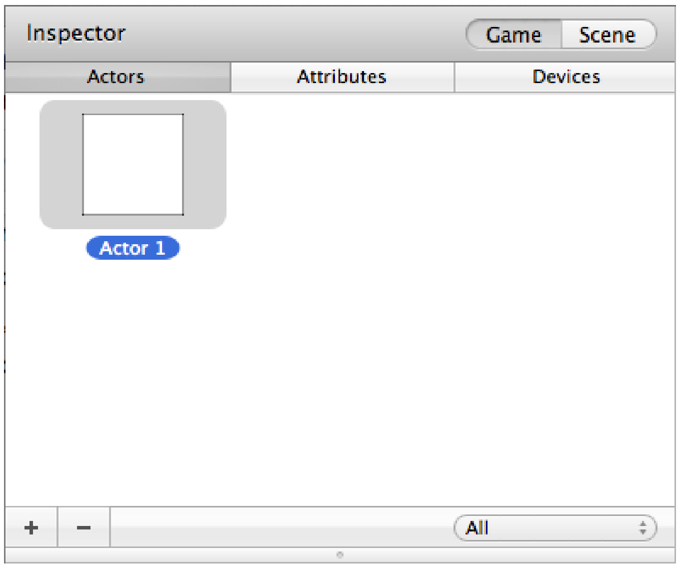


Fig. 10: A brand new Actor, ready for action.

We haven't attached an image, so it's just a blank white square, but that's fine. We can add it to the Scene by dragging and dropping it into the black box.

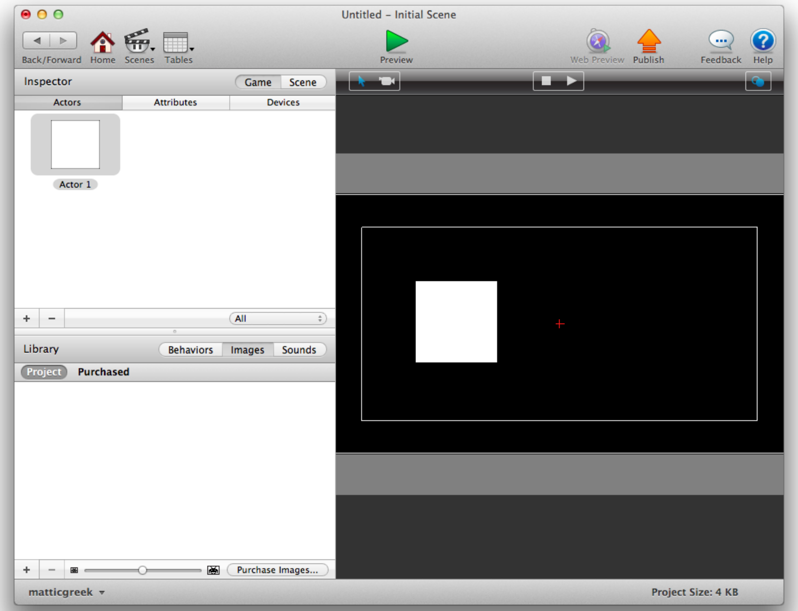


Fig. 11: We dragged and dropped Actor 1 directly into the Scene.

It's time to put the Behaviors tab in the Library to use. In order to make the square move, we have to assign behavior. To do that, we double-click on Actor 1.

But wait! We now have two white squares on-screen, both of which we can click on. What is their relation and which should we double-click? Well, if we double-click on the one in the Scene, it opens a view that refers at the top simply to "Actor 1." However, if we double-click on the one in the Inspector, we see reference to "Actor 1 (Prototype)."

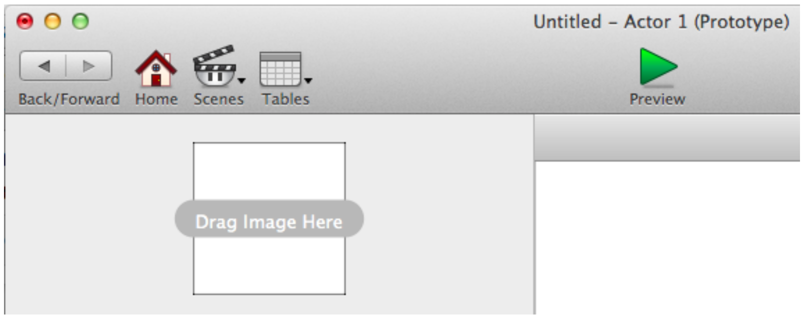
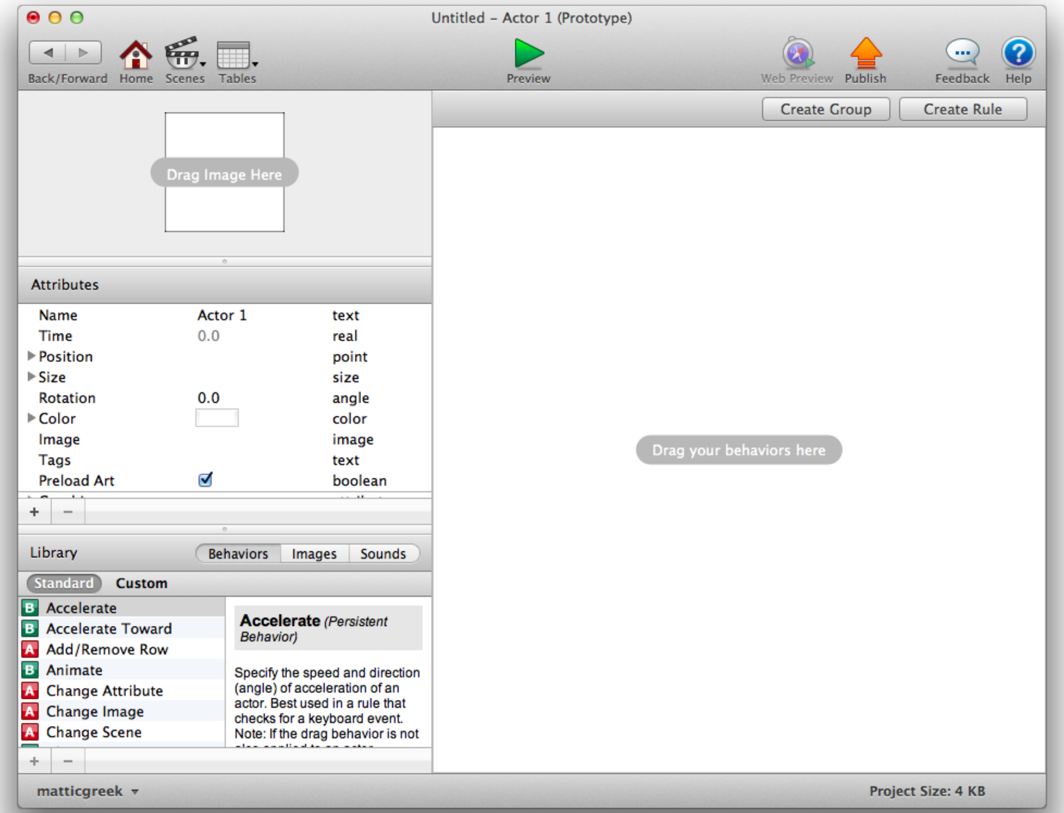


Fig. 12: We see at the top that this is the prototype of Actor 1.

From this we can deduce that the actor appearing in the Inspector functions like a mold, or the "class" in object-oriented programming. It doesn't have any existence of its own in our game, but we can use it to make a physical[30] copy that does go in the game. More to the point, we can make lots of copies. These copies, or "instances," all take their nature from the prototype. So, we can set the behavior for the prototype, then make many instances and they'll all behave the same. We double-click on Actor 1 in the Inspector.



Name	Actor 1	text
Time	0.0	real
▶ Position		point
▶ Size		size
Rotation	0.0	angle
▶ Color		color
Image		image
Tags		text
Preload Art	✓	boolean

Accelerate *(Persistent Behavior)*

Specify the speed and direction (angle) of acceleration of an actor. Best used in a rule that checks for a keyboard event. Note: If the drag behavior is not

Fig. 13: The "Actor" view for the "Actor 1" prototype.

On the left we see that the actor has its own attributes, and on the right is a blank white box. We know that we want to move with the press of a button, so we click "Create Rule" in the

[30] Well, relatively physical.

upper-right. This sets up a recurring check for a specific condition. We use the drop-down menus to check for if the right arrow key is pressed.

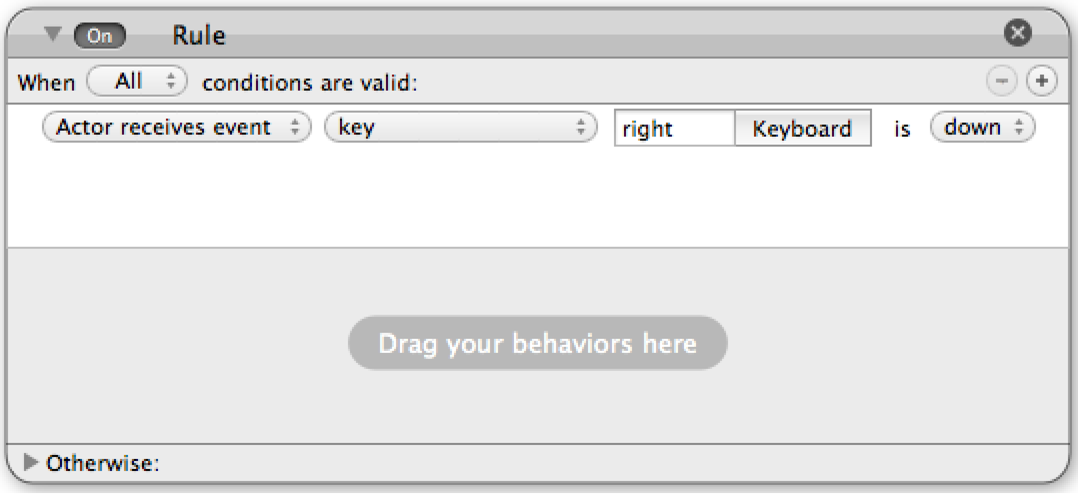
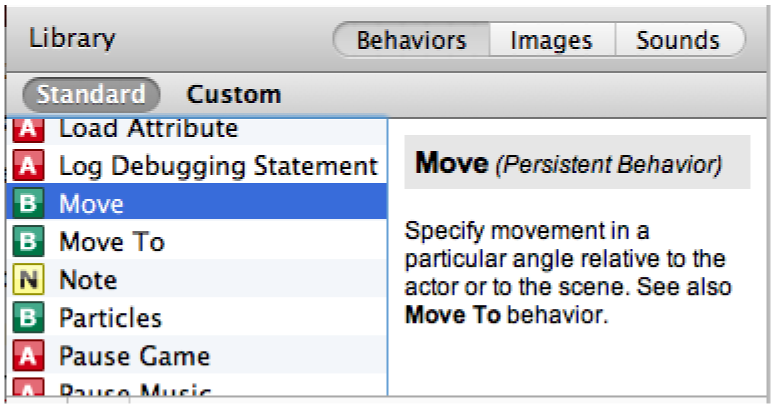

On
Rule
When All conditions are valid:
Actor receives event key right Keyboard is down
Drag your behaviors here
Otherwise:

Fig. 14: A condition check for if the right arrow key is pressed.

Now, in the Behaviors tab of the Library, we look through the options. The behaviors are alphabetized, and clicking on any one brings up a description on the right, two things that I really like. It's easy to see what things do. The occasionally-humorous descriptions even provide references to related behaviors--very classy! Scrolling down to the "M"s, we find "Move."

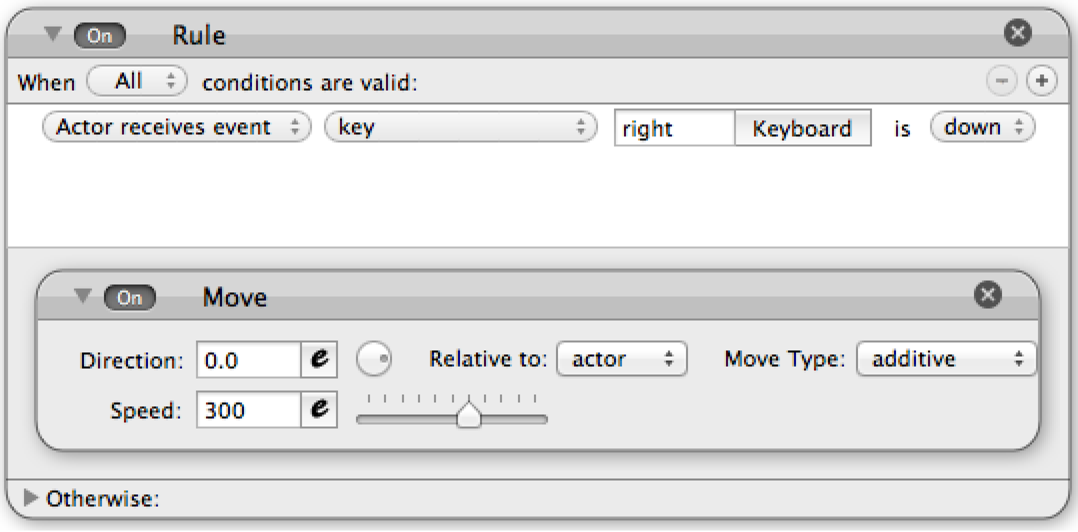


Move *(Persistent Behavior)*

Specify movement in a particular angle relative to the actor or to the scene. See also **Move To** behavior.

Fig. 15: The "Move" behavior, with handy description.

That sounds good, so we click on "Move" and drag it into the Rule we created.



On Rule

When All conditions are valid:

Actor receives event | key | right Keyboard | is | down

On Move

Direction: 0.0 | Relative to: actor | Move Type: additive

Speed: 300

Otherwise:

Fig. 16: Our behavior for the square specifies movement when the key is pressed.

The Move behavior has several parameters that we can set. Direction is specified as an angle, which the description warned us about and which might throw the geometry-uninclined for a loop, but we can also click the little nub inside the circle to the right of the Direction field and drag it to indicate the direction we want. Very nice. As it stands, the default values should be fine for our purpose, so we simply click the green Preview arrow at the top of the software and press the right arrow key.

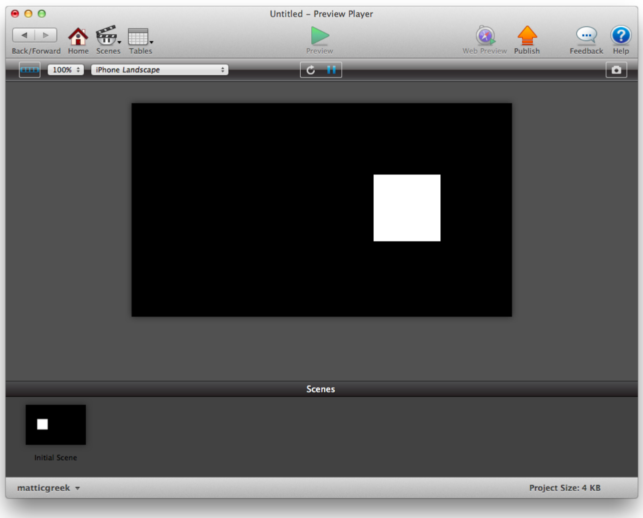


Fig. 17: Previewing the game. We moved the square to the right!

It works! The square slides smoothly to the right. Easy as pie. We could just as simply add controls to move in the other directions as well.

That, in a nutshell, is *GameSalad*. The interface is all drag-and-drop with (virtually) no programming, the aesthetic is very friendly and welcoming, and it's easy to use the many Behaviors to set up the prototypes for all the elements in the game, then drag and drop instances into the scene exactly where they should be.

We can even double-click on an instance to edit its individual parameters--if we wanted a specific enemy to move faster than the speed specified by the prototype, for example--which is in keeping with the relationship between class (default nature) and instance (specific nature) in programming. However, one has to be careful. Editing an individual instance severs its connection with the prototype, meaning that any changes made later to the prototype do not get passed on to the edited instance. This is true whether or not the change made to the prototype would actually conflict with the individualized change(s) to the instance.

The only way to reestablish the connection and apply new prototype behavior to an edited instance is to "revert [the latter] to prototype," thereby undoing any individualized changes. Or, we could make every subsequent change twice, once in the prototype and once in the instance--a tedious business. This is unlike true class-instance functionality, in which the instance retains its connection to the prototype but simply overrides any parameter in conflict.

Consequently, when using *GameSalad* it's wisest either to refrain from editing individual instances until all prototype behavior has been established, or to duplicate the prototype and create a separate actor prototype with the different behavior--for instance, enemyHorizontal vs. enemyVertical. This is somewhat like creating sub-classes, but not quite as robust. Still, within those limitations the system works well and indeed makes it very simple to create a game of some complexity with many moving parts.

VII. The Games We Make

So, what kind of game should we make? As mentioned before, *GameSalad* comes with several templates. We could choose any of them and build a game using the template as a starting point. Let's consider the options once again:

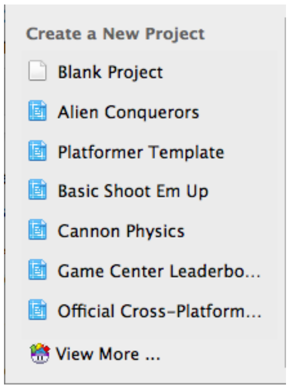


Fig. 18: The included templates in GameSalad.

The last two aren't games but rather demonstrate technical functionality in demand by the aggressive mobile developers we are expected to be. The "Cannon Physics" template may remind one of a certain popular game featuring irritated avians and is extremely impressive, but also a tech demo, not a game. Let's open "Alien Conquerors."

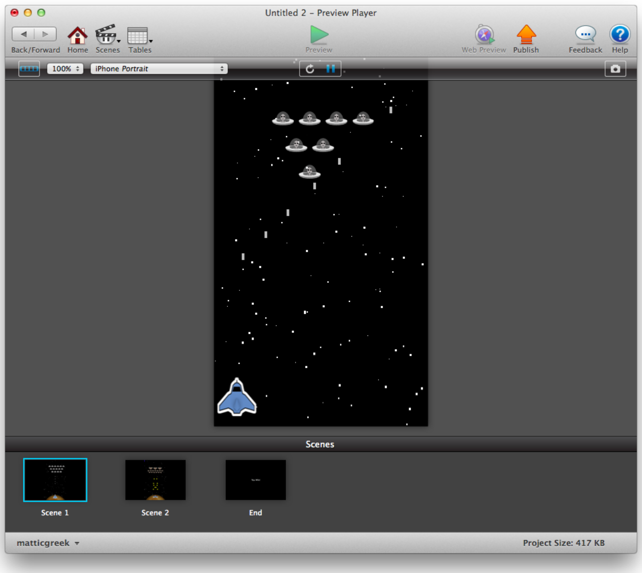


Fig. 19: The first game template, "Alien Conquerors."

OK, it's a space-shooter of the kind we've seen before; pretty standard beginner stuff. How about "Platformer Template"?

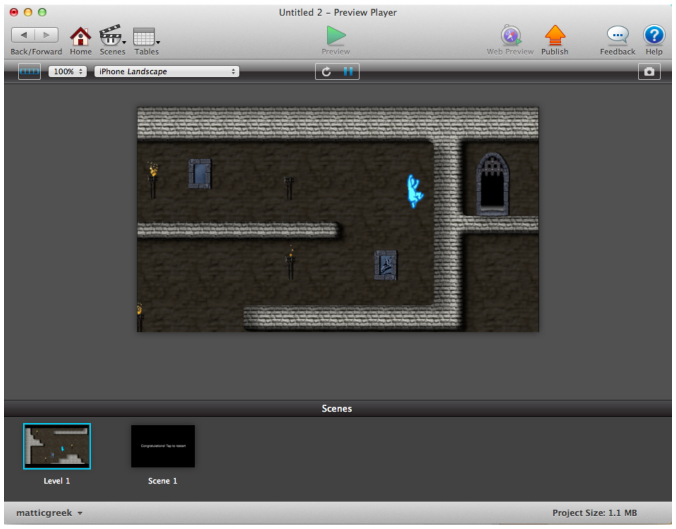


Fig. 20: The second game template, "Platformer Template."

It's a platformer, featuring the standard tropes of running and jumping and collecting things. What about "Basic Shoot Em Up"?

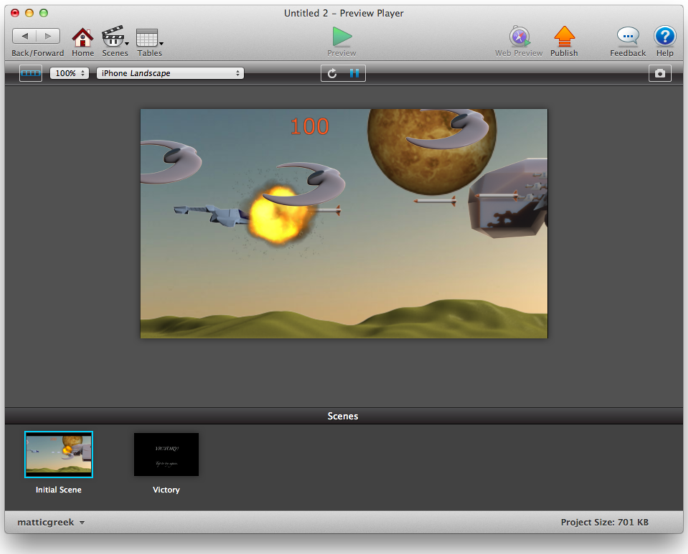


Fig. 21: The third game template, "Basic Shoot Em Up."

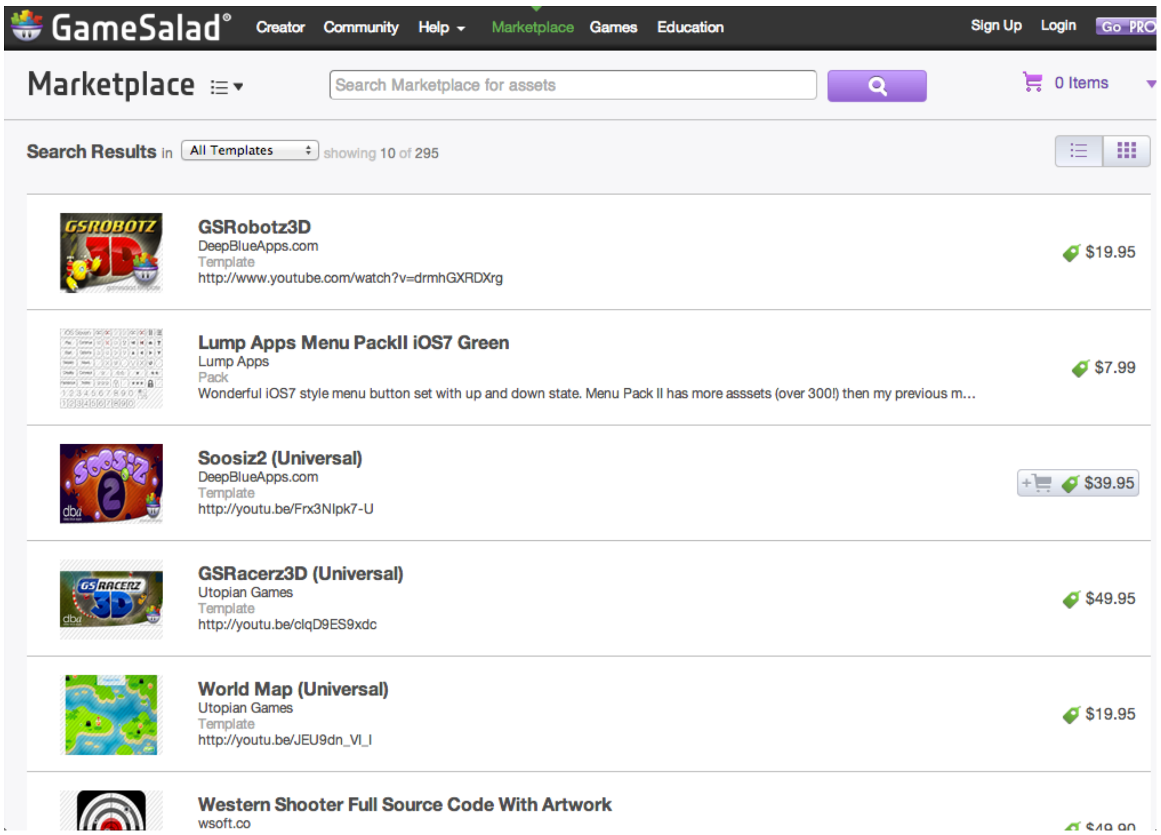
It’s...another shooter. Oh, but wait--this time we’re in atmosphere instead of space, and it’s horizontal now! So, to recap our template options, we have: 1) fly a ship and shoot things, 2) run, jump and collect things, and 3) fly a ship and shoot things.

I realize running, jumping and shooting are primal and they’ve formed the core of most game design up to this point. I further realize that they’re easy interactions to create and understand, and consequently are ideal for “My First Game (Genre).” But come on--*two* shooters? Couldn’t the third template at least have been a puzzle game or something to acknowledge gameplay diversity?

Here is further evidence of what I feel to be conservatism running through the *GameSalad* software. It is an attitude that says “listen, we know what is a ‘game’ and we know what people like and we know how to sell it to them and we know where to sell it to them, so just do things our way and be successful, OK?” And this is pragmatic, because indeed, shooters and jumping games remain very popular, and in-app purchases do make a lot of money, and mobile is the hot market. This is a product very much of its time, and that’s fine.

However, this attitude does not expand the medium, does not help video games to develop and mature and grow in all sorts of exciting and unpredictable ways. The tools are there in the software to create experimental work, to push the boundaries, to create games and experiences that move away from jumping and shooting, but *GameSalad* doesn’t go out of its way to encourage this.

Big props to *GameSalad* for the Arcade, because there the wilder stuff can shine through, if it’s good enough and people notice. But what if one wants to learn how someone else created something amazing in the software? Perhaps the community can fix the deficit in the variety of available templates? Well, yes and no. Here’s what happens when one clicks the “View More …” button underneath the template list:


GameSalad® Creator Community Help Marketplace Games Education Sign Up Login Go PRO

Marketplace

Search Marketplace for assets

0 Items

Search Results in All Templates showing 10 of 295

GSRobotz3D
DeepBlueApps.com
Template
http://www.youtube.com/watch?v=drmhGXRDXrg
$19.95

Lump Apps Menu PackII iOS7 Green
Lump Apps
Pack
Wonderful iOS7 style menu button set with up and down state. Menu Pack II has more asssets (over 300!) then my previous m...
$7.99

Soosiz2 (Universal)
DeepBlueApps.com
Template
http://youtu.be/Frx3NIpk7-U
$39.95

GSRacerz3D (Universal)
Utopian Games
Template
http://youtu.be/clqD9ES9xdc
$49.95

World Map (Universal)
Utopian Games
Template
http://youtu.be/JEU9dn_VI_I
$19.95

Western Shooter Full Source Code With Artwork
wsoft.co

Fig. 22: Lots of additional templates available...for a price.

Yes, it's a marketplace, the same one as we saw previously for getting additional images and sounds. There are loads of templates available, and some of them may be very unorthodox indeed, but they also cost money, and not a little money.

There is nothing fundamentally wrong with this--it incentivizes passionate creators to get involved with the community and rewards them for putting time and effort into creating content for others to use, assuming those others buy it. But it also creates a culture of commerce, not assistance. If the Marketplace were not there, people might still create templates and other content and make them available for others to use for free, but why would anyone make any such content available for free when they can sell it on the Marketplace instead? Of course there are still generous souls who record tutorials on YouTube and make resources available elsewhere at no cost, but the majority of the additional content is going to end up on the Marketplace, where there does not seem to be a way to tell how good the $50 template one is considering actually is prior to buying it. Spend money to learn how to use *GameSalad* better so that one can make money. Very practical.

As is readily apparent, I am not a big fan of the Marketplace model, but it's not unique to *GameSalad*--Unity does it too (with every bit as much appeal to me). I'm all in favor of people getting paid for their hard work, but I am not in favor of cheap cash-ins, which the lack of a review system on *GameSalad*'s Marketplace would seem to enable. Furthermore, it really bothers me that the templates included with the software are so limited, and that the way to get more is to spend a lot of money somewhat blindly. I fear that *GameSalad* users without deep pockets or the determination to figure out everything themselves may be excluded from a broader selection of ideas and implementations.

VIII. Dubious Design Decisions

During my time porting the Activate! games into *GameSalad*, the consequences of two questionable implementation decisions became clear, and I want to talk about those decisions. One is the default physics system, while the other is the lamentable absence of snap-to-grid.

VIIIa. Physics

Previously I commended *GameSalad* for its careful control over and presentation of information, which makes it easy to understand the software and how and why the behaviors work. The limitations of that ease of comprehension crystalize into clarity the first time one makes a wall, sets the avatar to collide with (i.e. bounce off) the wall, and then runs the avatar into the wall only to see the *wall* go flying off into the abyss.

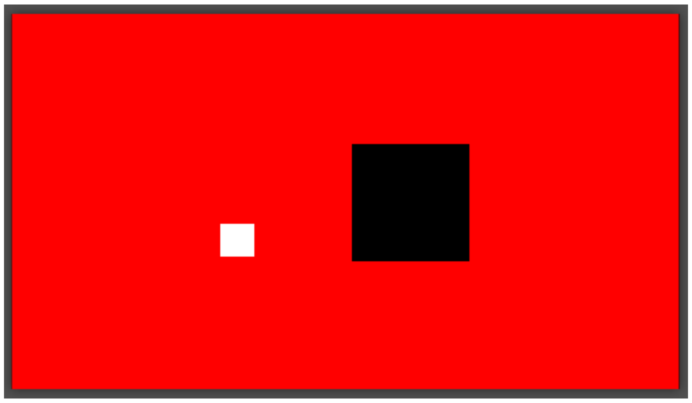


Fig. 23: About to run the avatar (white) into the wall (black), with which it should collide.

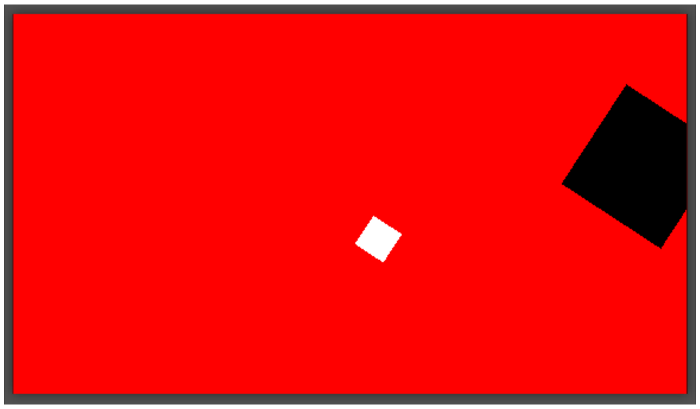


Fig. 24: Post-collision, the wall (black) flies off-screen; both wall and avatar (white) rotate.

The first time this happened, I was completely baffled, and I expect others would be, too. The reason it happened is that *GameSalad* has a robust physics system (as evidenced in the "Cannon Physics" template), and every single actor comes with physics properties already attached and in effect. Indeed, inside each actor's attribute list is a whole category called "Physics." Running the white square into the black one transferred force to the black one, sending it off on its personal journey of discovery.

In order to prevent the black square from moving, it's necessary to open its Physics attributes and uncheck the box for "Movable." And even after that, one may get unanticipated rotation on the white square from the collision unless one checks its box for "Fixed Rotation" (also under Physics). When the Collide behavior description says "bounce," it means "bounce" in that icky physics way, not "stop moving." That's great if one wants a ball to actually bounce off a surface, heading in the other direction, but it's not great if one wants, say, a character sliding on ice to collide with a wall and stop moving. To achieve that, one could either edit other, even more technical physics attributes like "friction" and "bounciness," or one could avoid "Collide" entirely and set up different behavior instead.

Physics systems are not easy to make and it's great that *GameSalad* integrates its own so completely. I question the decision to attach it to, and enable it for, every actor by default. Physics becomes something to opt out of rather than opt in to. Furthermore, if one wants to do different physics, one really has to work at it. Maybe if the software did a better job of explaining the system already in place it would be less of an issue, but the software does not, and indeed the "Platformer Template" does not really expose the physics settings that make the game work the way it does. One has to find them oneself. The rotation problem was especially baffling at first. I've gotten in the habit with each new project of disabling some or all of the physics settings for my actors, which seems like a lot of work to go through over and over again.

Making physics-based interaction the norm is another ideological stance, implying that all games should have physics[31] and that those physics should work roughly the same way from game to game. It's obnoxious (and opaque) enough to disable undesired physics properties that it may become tempting to design *around* them as a necessary evil. The physics system is so easy to leverage that it stops being a razzle-dazzle feature and becomes just another tool, but one that is very difficult to put back in the belt.

VIIIb. No Snap-to-Grid

This one hurts.

"Snap-to-grid" is a feature I first encountered in *GameMaker*. It divides up the game screen into a grid of squares or rectangles of specified size and draws that grid on top of the editing screen. It looks like this:

[31] In addition to in-app purchases and touchscreen controls.

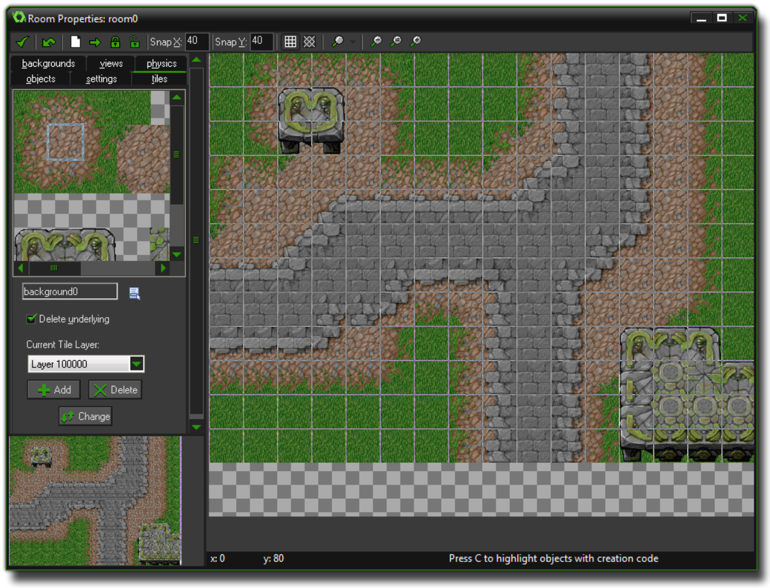


Fig. 25: The grid function in GameMaker.[32]

Once the grid is in place, one can choose to "snap" to it, meaning that as one drags objects around on the screen, their position snaps to align with the corner of the nearest grid cell. This makes it laughably easy to place objects with precision. It's even possible to align game elements with the grid during gameplay. For instance, one could make a chess game and make a moved piece line up perfectly with the board square. The feature is profoundly useful for map layout and quite useful for gameplay.

That's *GameMaker*; how about *GameSalad*? Alas, poor snap-to-grid, you are nowhere to be found. I have to take back what I said earlier about more resolution options for GameSalad Arcade being the most-needed feature. That crown goes to snap-to-grid, and it was no contest. It is *painful* to build any kind of elaborate scene layout with precision. Yes, one can drag and drop objects just where one wants them in a scene (just like in *GameMaker*), but the eye is not good at eyeballing true symmetry. So, if one wants to position everything with mathematical exactitude, one must double-click on each and every actor instance, calculate what its specific x and y coordinates should be, then manually set them. For every single thing in the carefully laid-out scene.

[32] http://docs.yoyogames.com/source/dadiospice/images/form_room_tiles.png

What if we need a line of perfectly-aligned trees across the top of the screen? We could create many instances and put them next to each other, then edit their position. Assuming we don't need them to behave individually though, we could also use image tiling, by stretching a single tree actor instance to be as wide as the screen and then setting the image to tile horizontally. That works really well! Except that, without snap-to-grid, we've got to guess what width is the right width so we don't end up with a fragmented tree image on the end because there wasn't enough space for another full copy. Or, we could mentally calculate how many pixels wide the actor should be, based on how many trees we want, and set it manually. Oh, and when one stretches an actor, it becomes less apparent where its x- and y-coordinates are located on its body. So, positioning the actor becomes even more frustrating.

Unless one is extremely patient, determined or insane, the absence of snap-to-grid means designing scenes that are less complex and less precisely laid out. Not all scenes need to be precise, obviously, but the fact that it is so hard to achieve precision promotes sloppier layouts. Yet the feature's absence also hurts gameplay in another way. To illustrate it, here is what happened when I tried to replicate one Activate! game's unique movement system. Warning: highly-technical discussion ahead.

"Lights Off" is a game about moving through a neighborhood while telling neighbors to save electricity and avoiding power-crazed "zombie" neighbors. You can bet it used *GameMaker*'s snap-to-grid. Every sprite in the game (whether tree, character or house) is 32x32 pixels, and the room (i.e. scene) that makes up each neighborhood is divided into a grid of squares, each sized 32x32. Paths are tight, no space is wasted,and everything is exactly where it needs to be, which is possible because the avatar's movement is aligned to the grid. Pressing the movement key moves the avatar one square; holding it keeps the avatar moving; either way, the avatar's position always aligns (i.e. snaps) to the nearest grid square upon key release.

Recreating this in *GameSalad*...did not go well. Recall that no space was wasted in the original level layouts. That meant that every path was 32 pixels wide, exactly as wide as the avatar. In theory, this could work perfectly well in *GameSalad*. The problem is that the collision detection is very exact. In order for the 32x32px avatar to move into a 32px-wide path, the two have to be *perfectly* aligned. If the avatar is one pixel off, it will collide with the edge of the path and refuse to move. The align-to-grid behavior in *GameMaker* took care of this, but there is no such assist here.

What are the options for movement? There is the Move behavior, which takes a direction and a speed. That seems like disaster because it would be almost impossible to line up the avatar with a given path. There is the Move To behavior, which specifies a coordinate point and a speed. That has more potential--we could set the avatar to move 32 units in the direction of movement. The problem is that it is still movement with a speed, i.e. crossing a distance, and it could be interrupted by an obstacle or an early key release. As soon as the avatar moves anything less than 32 units and stops, it's out of alignment with the imaginary grid, and we have the same problem. And if we set "run to completion," which requires the movement to complete the 32 units, we lose control of the avatar upon collision while it struggles to complete what's left of the motion but can't because of the obstacle.

Furthermore, there's the issue of enabling continuous movement by holding the key. "Move To" is not a repeating behavior, which means it occurs once every time it's cued. Pressing and releasing a key is considered a single event by *GameSalad*, so it only cues Move To once per key press, regardless of how long one holds the key down. I tried various stratagems to trick the system but none of them worked. So even if the alignment weren't a problem, it would still be necessary to continuously tap the movement key, which is tiresome.

There is "Change Attribute"--we could change the position to be 32 units in the direction of movement, which is instantaneous. But that circumvents the collision detection; the avatar can move right through obstacles! So far nothing is satisfactory.

Permit me to describe how one would achieve the goal in code. In pseudo-code (i.e. normal language), it would look like this:

If the position in the axis of movement is not a multiple of 32, add 1 until it is.

In actual code, it might look like this (for positive movement in the y-axis):

```
temp.y = pos.y;                 // Copy the current y-position into temp.y.
diff.y = 0;                     // Set a separate, new variable equal to 0.
if ( temp.y < 32 ) {            // If temp.y is less than 32...
    temp.y = temp.y + 1;        // ...Add 1.
    diff.y = diff.y + 1;        // ...Also add 1 to diff.y (started at 0).
} else if ( temp.y > 32 ) {     // But if temp.y is greater than 32...
    temp.y = temp.y - 32;       // ...Subtract 32.
} else {                        // But if temp.y is equal to 32...
    pos.y = pos.y + diff.y;     // ...Add the value of diff.y to pos.y.
}
```

This simply calculates the difference (represented as “diff.y”) between the current y-position and the next multiple of 32, and increases the y-position by that much.[33]

While there’s not support for writing code directly in *GameSalad*, one can create and manipulate variables, so I tried to implement the above code as a series of rules and just couldn’t get it to work. It’s possible I made some mistake and this sort of approach *could* work, but there’s no doubt that it fights against a system not designed for something like this.

In the end, I gave up on implementing the grid movement. Instead, I put the Move behavior back in place, and then I redesigned the levels to feature wider paths. That meant substantially revising the layouts, with all the tedious manual adjustment of position and size previously described. Ultimately, a similar game, not the same game.

This is an unusual case. But grid alignment is a staple of certain genres of games, and snap-to-grid is not a niche or minor feature. Its absence creates real design challenges, however closely they might mirror those from my case study. It’s worth noting that I did some research

[33] The code does assume the position always remains an integer. It would need to be adjusted for floats (i.e. decimals).

into grid movement in *GameSalad* and discovered an example using tables, which are another feature. I haven't talked about tables so far in this paper, and the reason is that creating a table in *GameSalad* yields this:

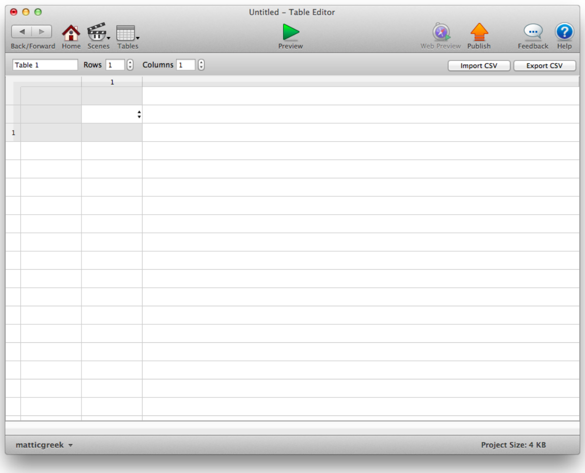


Fig. 26: A blank table in GameSalad*. Who's ready for spreadsheet fun?*

Yes, it's a spreadsheet. "Table," indeed. Tables are required to do things like manage in-app purchases but they can also be used for advanced calculations like those that might be necessary for grid movement. I didn't use them though because come on, look at that spreadsheet. How unintuitive can you get? This is contrary to the spirit of the rest of the software. To use a table, one has to know what one is doing, and to use it to do something like grid movement, one has to be an advanced user and/or have some idea of how to code.

So, if one really wanted grid movement, and really wanted to use *GameSalad*, maybe it would be worth buying a template, if there is one for that. But if the software had snap-to-grid, this wouldn't be necessary. The lack of snap-to-grid is a critical weakness.

IX. Conclusion

One of the games consistently featured at the top of GameSalad Arcade is called "CheeseMan Free,"[34] an utterly shameless--and openly admitted--clone of *Super Meat Boy*, and you know what? It's pretty good, quite impressive technically and aesthetically, and free

[34] http://arcade.gamesalad.com/game/60150

and easy to play in any web browser. And it is not a space shooter or generic platformer (if you have to steal, steal from the best). Clearly it is possible to craft impressive productions with the software.

GameSalad manages to be approachable while offering an impressive level of sophistication. The simple elegance of the UI and the friendly intuitiveness of almost everything make *GameSalad* a fantastic learning tool.

The publishing features are powerful and it's easy to put a game on the web. There's no question this software is primed to help people not only create but also distribute their games and even monetize them, especially in the lucrative mobile market. The community focus and Marketplace feature encourage people to create and share and earn income even from creative work other than finished games. *GameSalad* is very much a timely product. It's robust, capable, and clear-eyed about its priorities.

That same clarity makes me anxious about what kind of game creation it enables. The relentless focus on mobile publishing and in-app purchases, on tried-and-true game mechanics and standardized platforms, pushes against other ways of thinking and doing. By locking most additional pre-made content behind the Marketplace paywall, it discourages people from checking out and getting inspired by other people's work, except through the Arcade, which imposes its own restriction through the single allowed (and perplexingly small) game screen size. Finally, the imposition of physics and the unforgivable absence of snap-to-grid mean too much fighting against the system for no good reason.

In the end, I commend *GameSalad* for achieving its goal of being an easy-to-use tool that makes game development accessible to many more people, and I hope that as it continues to improve it will address its limitations. If it does not, I can only hope that those who learn with it will then move beyond it to other software with a broader mindset and more powerful tools.