SDB Level Design

Last Updated on March 14, 2005, for Shotgun Debugger 0.7
Document By Chris DeLeon

Table of Contents

Table of Contents 1
Intro 2
- How Reading This Will Lead to Better Levels 2
- What This Is, What This Isn't 2
- Communicate With Your Team! 3

Level Philosophies 3
- What is Level Design Philosophy 3
- Architect's Design 3
- Fireman's Design 4
- Curiosity Lure 4
- Item Lure 5
- Arena Traps 5
- Puzzle Based 5
- Creatively Linear 6
- Hybrids 6

Rapid Prototyping Patterns 7
- Node Network 7
- Division of Space 7
- Pipelining 7
- Mimicking 8
- Blueprint 8
- Area Sketching 8
- Hybrid 9

Element Tutorials 9
- Negative Space Design 9
- Chairs, Tables, and Prefabs, Oh My! 10
- Trigger Systems 10
- Interior Decorating – Patterns in Floors and Walls 10
- AI Pathing 11
- Unexplored Area Masking 11
- Fancy Lighting 11
- Taking Advantage of Layer Naming Flexibility 12

Common Inkscape Errors & Workarounds 12
- Boolean Across Layers 12
- Group Transformation Errors 12
- Scaling Inaccuracies 12

Golden Rules of Map Design 13
- Create, Create, Create 13
- Decide Early On If It's a Keeper 13
- Complete at Least 1 Map a Week 13

Intro

How Reading This Will Lead to Better Levels

Odds are, you already have at least a few raw ideas of your own for levels. You've played games before; you have your own concept of what's fun, and what makes a player quit half-way through a game. That's good – hang on to your ideas while you read this.

I have made maps for commercial games (dating back to *Doom*, *Descent*, and *Command & Conquer*), created maps for games I helped create (RPG, Bomberman Clone, bombing game, puzzle ball game), developed levels alongside mapmaking tools for games I finished independently (firefighter, brick-breaking, pac-man clone, 2 space shooters, fighting game), and have written tutorials online for game balance and scenario creation (*Perfect Dark*). From this, I have a reasonably well established base of experience to share. However, I'm not asking that you replace your current thinking with mine; I ask only that you add some of what's mentioned here to your repertoire, mix it with your best imaginings, and toss some of your thoughts back to me. It's one more way to look at things, and we're all on the same team. We can only benefit from helping one another create better levels!

What This Is, What This Isn't

This *is not* a technical specification on how to get a level drawn and compiled. That's what Matt Sarnoff's trusty level spec documents are for. I assume that you have already read Matt's technical specifications, and have familiarized yourself with Inkscape and svg2vl. If you have not yet done so, here is what you'll need to brush up on before getting started:

Design Doc	http://gcsociety.sp.cs.cmu.edu/~frenzy/design.pdf
Main Level Spec Doc	http://gcsociety.sp.cs.cmu.edu/~frenzy/leveldesign.pdf
0.3 Level Spec Additions	http://gcsociety.sp.cs.cmu.edu/~frenzy/leveldesign-0.3.pdf
0.61 Level Spec Update	http://gcsociety.sp.cs.cmu.edu/~frenzy/leveldesign-0.61.html

This *is* a document about designing a fair, engaging experience for the player through your levels. If Matt's documents tell you about the brushes and paints he's providing, then my document is intended to provide instructions on putting those instruments to good use to paint something others will want to look at again and again. Knowing how to put paint on a canvas, and being a painter, are two very different things.

Communicate With Your Team!

Matt (msarnoff), as the engine programmer and project leader, can answer any technical question you have. I (cdeleon) am the lead level designer, and can clear up any uncertainty about level development. John (jnesky) is in charge of art assets, so he knows what color the player's clothing is, and what style the robots are. Any SDB question you may have, one of us can answer it, so halting progress because of an unanswered question isn't an excuse. Post on the message board (board.gamecreation.org), and one of us will promptly address the question.

Level Philosophies

What is Level Design Philosophy

Every decision in a level's design is a conscious act by the level designer.

Levels aren't made by placing walls; levels are made by planning. Once a level developer is accustomed to the toolset at hand, and the underlying game engine, emphasis shifts from placing "walls" to placing "rooms," and from placing "enemies' to designing "situations."

Odds are, you've never played more than one published title that used the same underlying level design philosophy. What works in one game, given its AI, weapons set and player interface generally does not translate well to another title in the same genre. *Goldeneye N64* levels make poor *Doom* levels; *Doom* levels make poor *Unreal Tournament* levels; *Mario* levels wouldn't work for *Sonic* and *Sonic* levels wouldn't work for *Mario.*

Why bother with level design philosophy at all? Why not just make maps that are different every time? Just like there are conflicting philosophies that don't work well between games, for every game there are some overriding points that can be kept in mind to make the most of the engine. Here we'll identify and briefly analyze a variety of approaches to level design, to ease communication between the level team about what works best in the SDB engine.

Architect's Design

Some games focus on environmental realism, to the point that most of the levels are designed independently of the gameplay experience (ex. *Goldeneye N64*, *Hitman*). For these games, most of rooms, hallways, and open areas are laid out without any thought regarding player start, ammo/health boxes, or enemy placement. An architect by training is most likely to design this type of map, so we'll call it the "Architect's Design." It provides a strong sense of

immersion when it's done well, since real buildings aren't laid out linearly for mission objectives, but it can make for awkward flow that confuses first time players.

As an action game, we can't risk confusing our players. If they don't know where to go next, they may get bored and decide to quit playing. For this reason, Architect's Design may be inappropriate for SDB.

Fireman's Design

Other titles focus on flow of action. *Halo* is the unmatched example of this design strategy, although a handful of recent WWII first person shooters have crossed this with the Architect's Design to compensate for the disarray of the battlefield. The player is rarely left wondering where to go next, since there are typically shots, yelling, and action taking place where s/he should go. We'll call this the "Fireman's Design," since it results in the player rushing from point to point to "put out fires." This requires a considerable amount of event scripting, and doesn't leave much of an opportunity for the player to rest.

This might sound ideal for an action game, but for SDB it would expose too much about the game's underlying reliance on trigger scripting and enemy placement. It hurts replay value by making interesting things happen predictably the second time around. Because we're aiming for arcade style gameplay, we want to keep the experience up to the player as much as possible. Additionally, it decreases game-intensity/believability by 'darkening' areas the player has already been – once an area is cleared, its lifeless forever after. All the bells and whistles stop when the last enemy dies.

Curiosity Lure

Some games lure the player around via exploration. Tomb Raider and Descent both relied at least in part on this "Curiosity Lure" (the player's left thinking "maybe *this* pathway leads to the exit?"). It lends itself to arbitrary map layouts, leaving the player wandering in cycles through corners for the next area to search through. Either the areas being searched repeatedly have enemies that die once and never return (yielding boring, gun-cold gameplay), or they respawn enemies at some interval taking a heavy toll on the helplessly lost player (frustrating, unfair, monotonous gameplay). Either way, with a few possible special exceptions like boss areas, Curiosity Lure is the wrong approach for SDB. If you find yourself thinking,

"The player will *eventually* try this door/hallway," then you'll want to step back and re-evaluate the level's overriding philosophy.

Item Lure

"Item Lure," for the sake of comparison, was frequently used by id Software for *Doom* and *Hexen*. Item Lure laces corners, hallways, side rooms, and action areas with cheap, low-value items of cumulative worth. +1% health, +1% armor, and small ammo packs are ideal for this. Besides giving you as the level designer an instrument of instruction to show the player where to go, it provides an immediate visual test for the player to mark off areas s/he's already visited. If there are no more items in a room with several branching hallway exits, the player only has to explore whichever hallways still have items to find new areas of the map. This approach is a good one to keep in mind for SDB levels, since it constantly rewards our player, and leads to most or all of a map's areas being explored in turn. Note that care needs to be taken in a map designed with the Item Lure to minimize the depth of dead ends, to avoid the player ever winding up stranded without cheap items as hints.

Arena Traps

Although *Painkiller* is among the few recent commercial games to exploit this design strategy, it has been around at least since the days of *Mega Man 1*. The concept behind "Arena Traps" is to have the player fight battle after battle in isolated architectures. This avoids player's using kill zones to take advantage of deterministic AI. It insinuates an overriding, malicious intelligence manipulating the player's world, and although this makes sense in the context of SDB, the stop and go style of spawn-centered gameplay is more organic and slow-paced than the average SDB level ought to be. Because the mission in SDB is escape, and not absolute destruction, the player should be able to choose with most enemies whether to run past or fully engage. In the name of preserving player freedom, SDB probably should avoid Arena Traps.

Puzzle Based

Jumps, keys, physics engine exploits, and remote switches or time trip wires dominate puzzle based games. It's rare to see an entirely puzzle based game anymore, but some degree of puzzle is more likely than ever to find its way into every game on the shelf. *Prince of Persia, Sands of Time* had a few undisguised puzzles in the story, as did *Quake II* and *Tomb Raider*. For an action game like SDB, we're more likely to incorporate action puzzles into the gameplay

itself, as was seen in the Crucial-Fix map (giant robot) where the player needed to disable a laser trap by pushing and detonating an exploding barrel, or destroying each colored turret wire to open access to the last part of the giant robot. Keep puzzle pieces local, and keep players entertained. Make puzzles far apart, and make the player angry. Distant triggers should have a clear visual indication of what they trigger, and the player should never be required to push an object more than ¼ a screen to its required destination.

Creatively Linear

When the player is stringed from one location to the next, but they feel like it's their idea each time, then the level design is Creatively Linear. Done right, this describes a map that plays linear but doesn't feel linear. *Max Payne* and *Half-Life* are the best-selling games with this structure. It can significantly detract from replayability, and can also raise some issues of the designer having more fun than the player; if the player doesn't get to decide where s/he will go next, and instead the level designer does, who's really playing? On the plus side, it typically means the player won't get lost, and the emphasis of the gameplay is on action or platform/key puzzles rather than exploration.

A non-linear level structure (like one with a centralized hub and many hallways) can quickly be turned linear by an excessive use of keys, remote door switches, and other order-of-play constraining mechanisms. The workaround to avoid this problem is to take a *Mega Man* boss weapon approach, so that although the order of action is up to the player, there's an ideal order that the expert player will take to optimize level performance. For example, instead of having a red key on the other side of the map before getting through a red door, the player could have some hint that there's an ultra powerful HEPA gun on the other side of the level, and know that there's a dangerous crowd of enemies on the other side of the red door. For these types of unenforced Creatively Linear level designs to work, the player must be given clear clues or signs of what is to be found in different directions – perhaps by imposing architecture, detailed red warning patterns on the floor, or puddles of damaging green goup.

Hybrids

Most commercial games don't follow any one formula. The most interesting level design philosophies are those that manage to take the elements that work from a variety of design angles. Don't be afraid to experiment, but when you do so, you'll probably want to test it in a

smaller map before you build it in as a crucial point of a larger map. Part of hybridizing your favorite level philosophies will come from mixing and matching rapid prototyping patterns, which leads naturally to the next section:

Rapid Prototyping Patterns

Node Network

The scientist's approach. Begin by drawing a few circles on a blank page to represent major rooms, areas, hubs or hallway intersections. Draw a few lines between these nodes, until everything is connected in at least 1 way. Next, put your mind to thinking about how you'll run the player through these loops – item lure? Strictly ordered key puzzles?

Advantage: Fast way to sketch out how your level can have complex interconnectedness, without holding yourself back to conventional orthogonal hallways or linear path cutting as you might be tempted to do in Inkscape.

Disadvantage: Leaves a LOT of unanswered questions about the level's theme, and the layout of rooms. This prototyping definitely should be used only in conjunction with another method, such as Pipelining or Area Sketching.

Division of Space

The builder's approach. Begin on a piece of paper with a rectangle, pentagon, or a more interesting shape, like a giant hand (*Doom*, map E3M2 did this), and then recursively divide that space into rooms.

Advantage: Can create a more realistic building space (no wasted area between rooms or halls). Doing it freehand on paper then transforming it into Inkscape reduces the tendency to work strictly on-grid.

Disadvantage: Leads to a map that is potentially too visually cluttered, particularly since the overhead camera allows for areas to be seen even when outside the player's reach. If the overriding shape is not broken creatively, it can also yield a map that feels too contrived.

Pipelining

The mechanic's approach. This can be done equally well in Inkscape, if you're sufficiently comfortable with its environment and tools, as it can on graph paper. Basically, form

the level out of hallways, placing important items and keys in intersections or corners. Go back and bloat certain areas of the pipe into room-sized combat areas, add a little decoration based on a given theme, and call it done.

Advantage: As fast, if not faster, than any other method, without creating trivially simple maps.

Disadvantage: Often results in an unrealistic environment, and it takes some practice and patience to identify which areas of the pipeline should be bloated.

Mimicking

The beginner's approach. Basically, steal from anything and everything you can. Copy areas and layouts out of buildings you've been in, attach them to movie sets you've admired, and connect them altogether with an order of events that worked in your favorite game. As long as you're mixing multiple sources, you can result in some fairly decent levels while learning from the pros along the way.

Advantage: Creates much better content than you could have on your own.

Disadvantage: Makes you feel dirty and guilty inside. Do it if you're looking for a way to learn as you go, but try to wean yourself of it a.s.a.p.

Blueprint

The architect's approach. This actually involves sketching out the area as if it really existed – where do pipes go? Electrical wires? How and why are different areas (locker rooms, restrooms, offices, closets) placed where they are in relation to everything else?

Advantage: This method leads to the most believable environments.

Disadvantage: Slow, slow, slow. If you rely on this entirely, you can easily spend 2 weeks working on the same map, which is a huge no-no with deadlines approaching.

Area Sketching

The artist's approach. This technique is very popular in the industry, and often shows up in collector edition released concept art. Make 3-D sketches of areas you'd like to fight in – from the player's perspective, or from a "security camera" ¾ angled perspective from a room's corner against the ceiling. How do obstacles fit together? From where will the enemies come at the player?

Advantage: Creates memorable areas, vastly more so than any other means of design.

Disadvantage: Requires a little art talent, and some random inspiration. Do this when you can, but don't expect it to work for every room of every level.

Hybrid

This is the game designer's approach. Unfortunately, it's also a lot less formulaic in how to perform it properly. Knowing some of the other prototyping methods are there to work from, and how to articulate the different aspects to team members, is essential to succeeding in hybrid level prototyping. As design schemes can be mixed, so too design prototyping methods can be mixed.

Element Tutorials

Negative Space Design

Begin by creating a large rectangle in the arch layer. This is the block of marble from which you will carve the entire level. From here, create any number of polygons within that rectangle using Boolean addition, subtraction, and intersection. Be careful to only perform Boolean operations on polygons in the same layer! Once the entire level is one large conglomerated polygon within the first bounding polygon, Boolean subtract it from the main block.

The final step is to identify any polygonal sections that are separated from the rest – isolated polygonal figures (independent spatially) within the same Inkscape polygon will cause weird wall errors once loaded in-game. To separate these individuals, blanket each within a rectangle, duplicate the main polygon, intersection the rectangle against the new duplicated layer, duplicate the newly cut out polygon, and subtract the duplicated polygon from the main polygonal piece.

Advantages: Creates high roofs/ceilings between rooms and halls, and makes it much easier to create complex architectures than can be made via simple wall placement.

Disadvantages: Requires fancy Boolean work to prevent the engine from rendering the entire level every frame. Also, some care should be taken to differentiate areas by their objects and floor patterns, since the wall heights and styles will wind up the same without more fancy workarounds.

Chairs, Tables, and Prefabs, Oh My!

Create the pieces of your pre-fab in a new layer, titled "archprefabs" or some other layer title starting with "arch". Set various heights for the different polygons (the "h" tag in the XML editor – *not* the "height" value which indicates the polygon's overhead dimensions!), such as table legs and table top. If a piece is not meant to have side walls, turn off the Stroke value for that layer in the Fill & Stroke property box (CTRL+SHIFT+F). Group all of these pieces together (CTRL+G), then set a "cd" (collision detection) tag for the group to "nn" for No collision with characters and No collision with bullets. Then create a new polygon, a simple rectangle, with "cd" of either "ny" (shoot over the top, like a stool) or "nn" (block bullets and people, like a refrigerator). Set this bounding box polygon to an alpha value of 0 for Stroke and Fill, but leave both turned on, and whalla – you have created an engine friendly complex pre-fab with minimal collision detection! Copy and paste these into your level to your heart's content. svg2vl has errors sometimes with elements moved as a group, so after moving each prefab duplication, completely degroup (CTRL+SHIFT+G several times) then regroup (CTRL+G) to refresh any accumulated group matrix transformations.

Trigger Systems

Any number from 1-63 can be set as a polygon or character's "trigger" value. This corresponds to which events are caused by that polygon's touching, or object's destruction. Note that enemies can also set off contact triggers. You can then set a door's "lock" tag to -1 (so that no key can open it) and it's "unlockwhen" tag and "openwhen" tag to the corresponding trigger number. If you'd like to create a tripwire explosive system, just arrange for a surface with a trigger tag, and matching "killwhen" tags on exploding barrels or invisible "exploder" object types. Read Matt's documentation from 0.61 for more information on trigger types and uses. Be creative with how your arrange your trigger setups – these are what will separate the gameplay of one level from the next!

Interior Decorating – Patterns in Floors and Walls

Remember that polygons in the floor don't do collision detection against objects or bullets. Take advantage of this lightweight processing freedom to create some elaborate floor patterns. Hexagonal tiles, bricks, and intricate star arrangements are as easy to paint on the floor as they are to create in Inkscape. Be sure to take full advantage of Inkscape's array functionality,

and make style changes by grouping them. Again, remember that if you move elements as a group, they should be ungrouped and regrouped at their destination to avoid svg2vl mistranslations.

AI Pathing

Create a new layer called "paths", and place small rectangles in it to create AI nodes. When the game starts up, all nodes that can see one another from center to center will be linked into a path network. Once the enemy has seen the player, or had a "wakewhen" trigger value disturbed, it will follow this highway to the player. Remember that (a.) path nodes must have line of sight with one another to connect, (b) chained line of sight to the player at any given time for enemies to find him/her, and (c) be placed as few and as far between as possible to avoid dense spiderwebs of interconnected visible nodes. Version 0.7 has functionality in the options menu to view path nodes, so you can check to make sure they link as you expect them to.

Unexplored Area Masking

The layer titled "masks" has polygons which will vanish when the player steps beneath them. This is used to conceal an area beyond the player's exploration, since otherwise the overhead camera allows the player to "see through walls" into other parts of the level. Generally, these should be used very rarely, and kept very simple. Overused they will annoy a player (by design, they obscure information right in front of the player's eyes until s/he walks into it), but underused they can take all the surprise out of a map's most interesting areas. Use them to cover boss areas, and perhaps to obscure a player's surroundings when a player starts… or perhaps to provide temporary rooftops on buildings/boxes when a player starts outside of them. Remember to set the height "h" tag to keep these on top of walls.

Fancy Lighting

Lighting is as simple as creating colored circles of varying opacity in the "lights" layer. However, making it look nice means tying the lighting into physical entities in the level. Lightposts, evenly-spaced wall lights, ceiling lamps, and glowing computer terminals can add a lot to a room's believability. Lights with mysterious sources need not be avoided entirely, but they should be kept to a minimum. For particularly dramatic areas, like tall-pillared entryways, consider placing false shadows on the ground in the floor layer as black shapes with low alpha values. Lines can be accurately traced out from a light source by creating a 1000 point star in

inkscape with long narrow points, to use as a dummy for guiding floor polygon creation, and then deleting that star before loading the map into svg2vl.

Taking Advantage of Layer Naming Flexibility

Matt has altered the engine to support suffixed names for the essential layers of a level. Every top-level layer label starting with “arch” will be put together in the game’s arch layer, so you can freely create “archnegative” and “archmainhall” layers, along with things like “archdecorations” or “archceilings”. These can help considerably in the level making process, since you can easily hide or unhide entire categories from your layers. Just remember to not perform any Boolean operations across different layers.

Common Inkscape Errors & Workarounds

Boolean Across Layers

Trying to add, subtract, intersect, or perform any other Boolean layer combination between layers will not raise an error or warning – it will instantly crash Inkscape to the desktop and cause you to lose any unsaved work. Make a habit of being aware of which layers you’re working with when doing Boolean operations, and save often!

Group Transformation Errors

Grouping (CTRL+G) elements and then moving them can often lead to translation errors with svg2vl. Solve this problem by ungrouping (CTRL+SHIFT+G several times, to also ungroup any subgroups) and regrouping (CTRL+G) once a group is moved to where you would like it to stay,

Scaling Inaccuracies

Scaling anything in Inkscape can case unpredictable placement and size errors in svg2vl, so avoid scaling operations whenever possible. Instead, operate on the polygon on a per-point basis, and simply relocate each point of the vertex to your desired location to simulate a change in scale.

Golden Rules of Map Design

Create, Create, Create

Never stop trying new things. The level engine supports features that neither Matt nor I have considered, if you'll only discover for yourself the right way to combine the properties and features that already exist. Design around themes we haven't considered. Design with methods I haven't discussed. Design, design, design – but keep your designs practical by compiling and playtesting them as you go.

Decide Early On If It's a Keeper

If a map is a stinker as a playable floor plan, then it won't be any less of a stinker once you've put a little lipstick on it. If you make a floor plan and decide that it's way out of whack in scaling, takes too long to run-through, feels too random, or feels too linear, nix it *before* you've spent 6 hours making it pretty. Try a different approach and mentality the next time around.

Complete at Least 1 Map a Week

No matter how many maps you have to go through in a given week, make sure that you have at least one map to show that's complete by each Friday. To be complete, it needs a clear beginning, middle, and end. It needs ammo and health, bad guys and lights, triggers and doors, some sort of coherent theme, and ideally a little touch of creativity that will set it apart from other levels being produced by the team. The only way to get better at making maps is to make maps, and to make maps is to finish them. At the end of this project cycle we need a finished game with playable maps, and if a couple of your earliest completed rough maps are the best we can include, they'll do more justice to the engine than no maps at all. Finish 1 map for the team by each Friday, and have it as a playable vl file in your folder on the gcsociety server. You'll be glad you did, when you start to see your map making abilities take off as you become an expert in the process!