Collision Detection and Response

Collision Detection

Checking for collisions in a video game is just like doing a search for a value in a given data structure. Instead of using a single value as a key, we are going to use the position and shape of an object together as a key to determine what other objects overlap the object being tested. One big difference here is that testing the equality of keys (testing if the shapes overlap rather than just testing values against each other) is much more computationally expensive as the complexity of your shapes increase. As in traditional data structures, you are going to want to select data structures and sorting schemes for your objects based on how they move and interact with each other to reduce complexity. The original dxframework puts all of its objects into a linked list, then sorts them by their position on the x-axis so it can throw out all objects after the first one in the list that is too far to the right to collide.

Determining whether two objects intersect is closely related to how they are represented graphically. The two different ways to represent objects on a computer screen are to use raster or vector graphics. Most three dimensional games use vector representations of objects and the graphics card rasterizes the image to be displayed on the computer screen. Many two-dimensional games that used to represent objects with raster graphics are now using hardware to render images. This allows for flexible video resolutions and scaling for camera zoom effects, but removes your ability to use rasterized images for determining overlap of objects. When working in three dimensions you use voxel graphics. Voxels are like pixels, only they are cubes that represent a volume, instead of squares that represent an area. In raster based graphics, you must compare every pixel or voxel of the overlapping space of both objects to determine exact collision. With vector graphics you end up comparing every line or plane. Both of these methods require lots of computation. The easiest way to simplify these comparisons is to enclose all of your objects into Axis Aligned Bounding Boxes (AABB). Determining intersection between two AABB's is probably the fastest of any of the geometric shapes used to represent objects. Once you determine that their AABB intersects another object, you can either treat it as a collision, or do more precise detection. This is referred to as the Broad Phase of your collision detection. If you throw out all of the precise detection and treat it as a collision, you need to choose AABB's that reduce the amount of false hits, which happens when the AABB's overlap, but the objects visually don't overlap. Most games minimize the amount of open space inside the AABB, and allow the extrusions to be uncollidable. If you do use the precise collisions, your boxes need to enclose the whole object, otherwise it's possible for the extrusions not to be detected. The more precise detections are part of your Narrow Phase Collision Detection. It is usually best to use multiple AABB's per object if a single AABB leaves too much space inside the box or leaves too much of the object outside the box. Circles and spheres are other common shapes used to enclose objects. The intersection test between circles or spheres is very simple, and you can get away without using a costly square root function if you don't need to know by how much they are penetrating. The biggest advantage with circular objects is that they be rotated by any angle and still retain their orientation. This way if your objects rotate, all you have to do is rotate the center points of the circles around the same point the object is rotated. You can get a similar effect with AABBs if they are squares, otherwise there is a good chance the AABBs won't represent the space the object fills anymore. To solve this we could rotate the boxes, but then they will simply be bounding boxes (BB) that aren't aligned, which are more difficult to check intersections of. With the clever use of mirroring you can at least get the cardinal directions of rotation of your objects without having to give up your AABBs. A strategy a lot of older three dimensional games use to simplify collisions is to limit the axes in which objects can rotate. A lot of games were being converted from their two dimensional predecessors which didn't use rotation at all, but now with characters being able to travel anywhere, they need to be able to go in any direction. Cylinders are very good for objects that don't need to change their pitch or roll and the detection algorithms wouldn't be much more complex than circles if they are aligned to the z-axis. If you need your objects to rotate on all axises, then OBBs are your best bet. Algorithms for determining if two arbitrary shapes without curves intersect is tricky but possible to do in real time if you use the above methods to throw out a lot of computation. It does require a lot of time to sort though all the special cases but creating a function that determines whether two objects intersect only needs to return a Boolean, so its possible to catch most if not all cases.

Most of the time, your objects will be anchored by a "hotspot" that represents the object's position. Hotspots are single points used to test intersections with other objects. All of the object's geometry will then be relative to that spot. Sometimes you can approximate complex objects with a series of hotspots. Collisions can be lost if other objects are small enough to pass between the hotspots, so they are only good for objects that don't need to collide with other objects made up of hotspots. The advantage they offer is that determining if a point is inside nearly any kind of shape is fairly easy. They are widely used in tile based engines to reduce the amount of tiles needed to be checked for collisions. In the olden days of Nintendo, doing more than four tile look-ups per collision pixel per frame for sixty frames per second was too slow. Another common use of hotspots is projectile creation. This way you can just check a frame of animation for a hotspot that represents a barrel and a position a bullet should be created when it is fired. This is useful for objects that can collide with the object that fire them. You have to make sure in that case that the bullet is created outside the collision bounds of the shooter. You can also make the bullet unable to collide with anything the first couple frames it comes into existence to represent the fact that they are coming out of a barrel and not hitting it.

Once you have a function that can detect an intersection between two objects, you need to start creating data structures for your objects. The most strait forward way is to stick them all into a single list and test every object against every other object for a collision. You can use a linked list or simply an array to implement it, depending on what operations need to be done to the lists. This way you are guaranteed to detect every collision that happens in your game world. Since you are checking all objects against one another, they can move around freely without needing to change any of the object positions in the list. The greatest drawback of this method is that it is an O(n^2) algorithm. There are no optimizations based on the properties of your objects. You can turn it into a O(n+m) algorithm by testing the first object with all objects after it in the list. Any object before it would already be tested in the previous search. You can optimize this algorithm further by sorting your objects by their x coordinate or their y coordinate depending on how they will be oriented. It may be possible to sort them by distance from the origin, or even distance squared from the origin to get rid of that square root function. If your objects are positioned along the x-axis, it is best to sort them by their x coordinate. Not as many object would be thrown out using the y-axis because there could be a large number far away to the left and right, but would still need to be tested because they are in range along the y-axis. If you have a game where the levels are long corridors running up and down it would be better to sort by the y-axis.

Most of the older console games and now the new hand held games use tile based systems for collisions against a static world. The way this is accomplished is by breaking space into cells of regular size. Then each cell will contain a different object that can't have any collideable parts outside of the cell. The easiest shape to use is an axis aligned square that is the same size as the cell. Older games only use squares, but more sophisticated games use slanted tiles and even curved surfaces to make the environment seem more natural. The newer games also use smaller tiles to make the environment seem more natural. The smaller the tiles are the more natural it will seem. Some games even make their tiles only one pixel. With a grid, we can easily map points to cells. Given an object's AABB, we can find the cell the upper left point is in, then find the one the lower right point is in, then scan all of the cells in between. Once we know the cells that the object intersects, we can run the detect collision function with the objects in those cells. The most powerful aspect of this method is that we now have an O(1) algorithm for testing collisions with static objects. The amount cells we test grows with the size of the object, but for any given object the number of cells that need to be tested is a constant. This means we can have our worlds be as big as memory allows without killing our processor. To move these objects around you will have to have them snap to the grid, and not allow more than one in a cell at a time. Another method is to remove the object from the cell and put it in your object list. When the object stops you can have it snap to the grid and put it in its new cell. If you want your objects not to be aligned with the cells, you need to allow objects to occupy multiple cells. So the mobility of these objects is reduced from this system. This algorithm scales to three dimensions very easily. It has been used in most isometric view games. It can also be effective in games rendered in 3d. When 3d first started getting popular, a lot of designers got caught in the idea that they had to use algorithms based on the data that renders the polygons. However games on the Playstation seemed to use the cell based algorithm more frequently. This method is also used to create games that use hexagonal cells. Any shape that will tessellate and allow you to map a point to a cell can be used with a little more computation. The most common way to implement such a system is to create a bitmap that bounds a single shape. It will contain one whole shape and the area not filled by it will be the adjacent cells. So really it uses a combination of rectangular cells and hexagonal cells. First it checks what rectangular cell the point is in then, it checks which hexagonal cell the point is in with the bitmap. Really bitmaps are just forms of grid based systems with each cell being one pixel. You could also take a more mathematical approach once you find the rectangle the point is in to determine which hexagon the point lies in. A similar system can be used to create isometric view games, but I find it very confusing to work in a space that is relative to the camera angle. I find it is easier to work in space that is oriented to the origin and just project object positions to an isometric view. This type of game can get very complicated because when designing your data structures for objects you have to make sure that the structures can be traversed quickly to check collisions and be traversed in the correct order quickly for rendering.

A lot of two dimensional games are given a three dimensional look by making graphics for the tiles that appear to be 3d, but collisions are handled the same way as if they were 2d. A lot of overhead view games use graphics that make the view appear to be more at an angle rather than directly overhead. It is usually easier to add details to characters this way because now you can see their whole bodies instead of just the top of their heads. The problem we are faced with now is how to choose bounding boxes. We could choose them so that they bound the whole image, but collisions won't look quite right. What will happen is when the head of one character touches the feet of another a collision will occur. Really the two characters haven't come in contact, but the first character's head is now obstructing the view of the second thus the images intersect creating a collision. A lot of older games simply used this method. A most sophisticated way to handle this problem is to bound the lower part of your characters. This way if the same intersection happened there would be no collision and it would look as though the first character's head is just in front. There is a good chance you will need to alter your sprite lists by sorting them by the y-axis to have them render correctly, otherwise it would look like the character behind was in front. This is where your rendering algorithms start affecting your collision algorithms, unless your hardware is fast enough to sort the list twice in one frame. Tiles need to be handled slightly differently. Tiles that represent the upper part of objects need to be rendered after the sprite layer is rendered, and tiles that represent the bottom part of objects need to be rendered before the sprite layer. This way objects behind the tile objects will be obstructed by the object that is too tall to see over. Similar techniques are used for side view games. There are many variations to this technique but most seem to stick to 2d collision algorithms. There are games that are rendered in 2d, but the collisions happen in 3d. Some of them could even be converted to 3d simply by changing how the game is rendered to the screen.

Binary Space Partition trees were one of the first data structures used in PC games to allow for arbitrary level design. Now you could make level geometry without being constrained to cells or any specific orientation. The way this works in two dimensions is all lines get stored in a tree. The first line gets stored as the root node. Then based on whether the next line is behind or in front of the first it is placed in the tree. Let's say that if the line is in front of root, it is placed in the right child node, if it is behind it is placed in the left child node. This algorithm is carried out recursively until all lines are placed. If a line crosses another, it has to be split in to two lines that are put to the left and right nodes. For three dimensional geometry, instead of lines planes are used. This structure is good for surface removal in rendering and also speeds up ray casting. It is also easy to tag moveable objects in the tree to determine which lines or planes they potentially collided with. Its main drawback is that the tree has to be made ahead of time because it is too slow to create on the fly. This means all the geometry in the tree has to stay put. Another approach using trees is to go with octrees or quad trees. As the name suggests, octrees are trees where each node has eight children. The first step to using octrees is to bound your entire world into an AABB. Then its children will be eight AABBs that were created by cutting the original AABB in half along every axis. Every node contains a list of pointers to the objects that occupy it. Your tree can go as deep as necessary, eventually coming to the point where there is no more than one object per leaf node. Now if your object doesn't intersect a node box, then you can throw out all its children. Objects are slightly more mobile here, but movement requires a lot of shuffling of objects to one list to another. Quad trees behave the same way, only space is split up into four sections instead of eight.

There are often times when you want to sweep space to find the first object that a vector, sphere or box intersects. Most first person shooter games use a ray cast to figure out what object a bullet hits. Because bullets move so fast, it is usually best to do a single ray cast during the frame the bullet was fired. Doing this is similar to doing an intersection test between two objects. The difference is we have a ray with a starting point and ending point. The first step is to find all of the objects the ray intersects disregarding the direction the ray is going. Then from the direction, we have to find which of the objects intersected the ray runs into first. To do fast ray casts you need to have your space partitioned into tree structures or grids otherwise you will have to iterate though every object in the game world. Ray casts are often used for artificial intelligence algorithms. When your computer controlled characters are deciding where to go, it's good to know if a wall obstructs their path or if they can even see their targets to pursue them. Ray casts will be very useful when you start moving your objects around.

Most games use a combination of the above methods. The most common combination of older console games is a grid for a tile based world and a list of objects. Hot spots are used to detect collisions with tiles and AABB's are used to detect collisions with all of the other objects. This method is widely used in the old console systems and many hand held systems because the hardware is designed to use them. The Super Nintendo was able to render a few tile lairs at once along with a sprite layer. Some times the layers were used simply for graphical effects but they were also used to add different layers of objects on one another. A slight variation of this for large worlds is to break up the world into sectors, and have each sector hold a list of every object in that sector. This way objects only need to be checked against objects that are in the same sector. You may get multiple collisions for a pair of objects in one frame if they are in two sectors that are the same if you are not careful. Instead of using a tile grid for your world you can use a BSP tree or octree if you have more complex worlds. A lot of effects can be created by shuffling your objects around in your data structures. If you have a bridge that is made out of tiles and you want it to break and fall, you can remove the tiles from the grid and put them in your object list for them to move around. Once they hit the ground and again come to rest, you can put them back in the grid in their new positions.

Collision Response

Once you decide what types of shapes and space partitioning you want to work with, you will have to choose a frequency for which collisions are tested for. If you choose a frequency that is to large, your game will slow down, and if you pick one that is too small you run the risk of objects passing entirely though each other in between collision tests. You can avoid this problem by using sweeps, which are like ray casts, but you are sweeping two objects though space to determine where they collide, but you can't always guarantee a constant player input frequency with this method. The solution for sweeping multiple objects beyond two becomes extremely complex, so most games use the prior solution. It becomes even more complex once you add in integration for rotating and moving the objects though space. The most you would want to do is perform sweeps with static objects. The frame rate for updating the screen can be totally independent of what some call the simulation frame rate. Quake and similar games broke the tradition of keeping the rates at the same frequency because before graphics cards, rendering was a big bottle neck. Although the frame rate was no longer constant, the simulation step remained constant in Quake. If the game execution was falling behind, it would just skip a rendering frame and update the simulation multiple steps. This concept grew when games started being played over the internet. Quake III's simulation steps are done at 30 frames per second, and allows the graphics engine to interpolate between them on the client side. This allows for arbitrary render frame rates depending on how fast your computer is while maintaining consistency in the physics simulation.

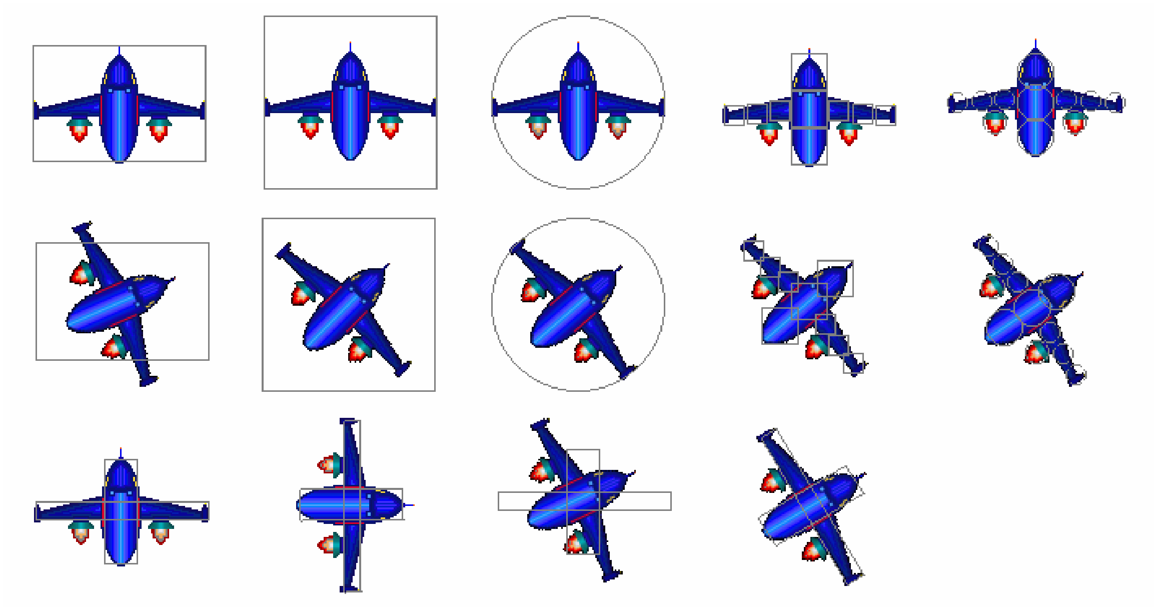
Now that you know your objects won't pass though each other, or at least reduce the chance they will, you need to decide how the objects are going to react once a collision is detected. The response can range from simply subtracting from an objects life value, or giving the object the exact momentum, velocity and/or spin it would have acquired if it was a real object in space. The latter has become much more common now that third party rigid body simulators such as Havok are being created to speed up development of games. However full rigid body simulators aren't always necessary, it can be much easier to track error and reduce physical anomalies in your game engine using less accurate approximations. Consistency is an extremely important element in games, without it you run the risk of creating a confusing and frustrating experience for the player that could have otherwise been an enjoyable gamming experience. The most difficult and costly computation of a physics engine is finding exactly how and where objects intersect. Many early and even recent games avoid this problem all together by simply giving objects that collide a velocity vector in the opposite direction. This is an example of soft penetration constraints. You let the objects overlap, but by applying forces you make sure they don't intersect for long, thus making it appear they never penetrated. Your other option is applying hard penetration constraints. When two objects overlap, you move them out of each other by finding the exact points they collide at, and then applying whatever forces necessary. This method is used more frequently for static environments to avoid having your objects "jitter" around as they are constantly knocked outside the floor or wall.

There are several ways to determine the point or points of contact. One common approximation is to move them out by the least distance they are penetrating. This calculation is faster than calculating the actual vector the object needs to be moved along to find the exact point of collision. With this method you either have to move one object along this vector, or move both half the distance in opposite directions. If you are making sure the object doesn't penetrate your static environment you need to use the first option, or you player could potentially move the terrain around. The second evens things out a bit by deriving the same solution regardless of which collision is detected first. If there are multiple players in the game, this would make sure none had a priority over any other. The way most systems work is by iterating though every pair of objects a collision is detected for, so objects will move around between collision responses. This means that the solution will be dependent on the order of objects in our list. Unless you want to add a high end numerical system, you will have to let your solvers resolve the collisions in what ever order the objects happen to be in. The list could be sorted or unsorted but in the end result you can't tell the difference unless there are large amount of objects colliding at once. This isn't a problem when using soft penetration constraints because objects aren't getting moved around during the collision response, just their forces are being altered. One way to get accurate results using hard penetration constraints is a process called bisection. Suppose a simulation step starts at time t0, and ends at t1. If at time t1 we detect a collision between any object, we know at some delta time past t0 a collision occurred. To find that delta t quickly, you need to keep dividing the time interval by half until the distance between the colliding points is some small threshold epsilon away from each other or interpenetrating each other. Once no objects are penetrating any longer, we can react to the first collision detected by updating the momentum of each object and continuing the simulation normally until you reach t1. This process could also be used for the list of object pairs that are colliding on one frame, but with the same anomalies that arise from the order in which the collisions are responded to.

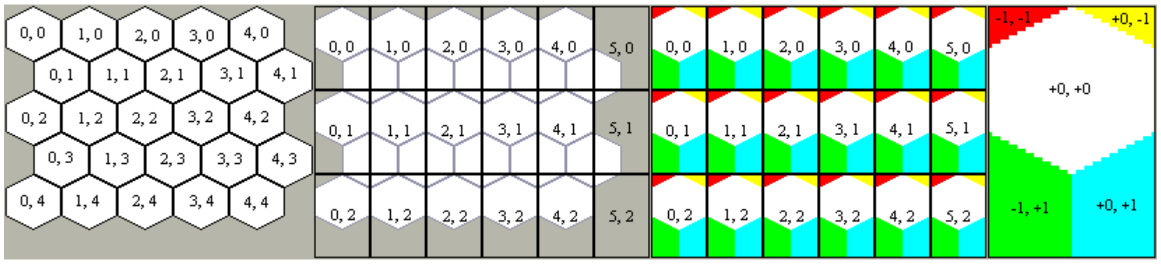
Even if you derive perfect penetration constraints between all of your objects, you still have to maintain conservation of momentum. From Newton's Cradle, we know that energy could potentially be transferred from one object to any other object in your game world. This could create a problem if you are using space partitioning and you don't have time to run though your whole list of objects for every other object in the world. Depending on your game, you may be able to get away with only transferring energy between objects that are in immediate contact. This would be easy to implement if your system returns a list of object pairs that have collided with one another. Maintaining energy conservation is much more important if you are designing a fluid simulator, where all of the fluid particles at the bottom of a mass of liquid will be affected by all of the particles above and around it. A good way to approximate those conditions is with the use of a kernel function, or a function that determines which particles around any given particle should affect that particle. For example you may only have particles be affected by other particles within a certain range. Ideally you would want an infinite kernel, which includes every other particle in the universe.

Many console tile based engines avoid complex penetration constraints all together. Collisions between sprites and use entirely soft constraints, and collisions with tiles are done with hotspots. You can get away with using about eight hotspots to define your character, and depending on which one collides with a tile, you know which way the object should be moved out of the tile. Hotspots typically defined are three for the left and right side, one for the top, bottom and middle. Then two are defined for the top of the head and feet. If your frequency for collision tests is small enough, you will know if a right side hotspot collides with a tile, you need to move your character outside to the left. Or you could make it a bit more robust and store where the character was last, and move them outside the tile along the x-axis. The third type of object commonly used is a moving platform. This is the only object that hard penetration constraints are enforced for all other moving objects, except other moving platforms. The problem that arises with multiple objects of this type is when you detect a collision, it's not always clear which object should be moved out of which. But using the policy of just moving all normal objects outside of the moving platforms is very simple. Your moving object could also be an entire tile map. Some games have moving trains you run on, crushing ceilings, and many other effects using moving tile maps. This method could be extended using object priorities, and have objects of lower priority be moved out of objects of higher priorities. Another advantage to this approach is for object oriented designs is you can ignore collisions with the different hotspots to make different types of tiles. This will let you know what side the object entered the tile. A common type of tile used is one where the player can jump up though it, but lands on it when falling from above. This can give the game a pseudo 3d feel. Colliding with diagonal tiles can be made easier with this approach by making special tiles around the slanted tile to notify the object of its presence.

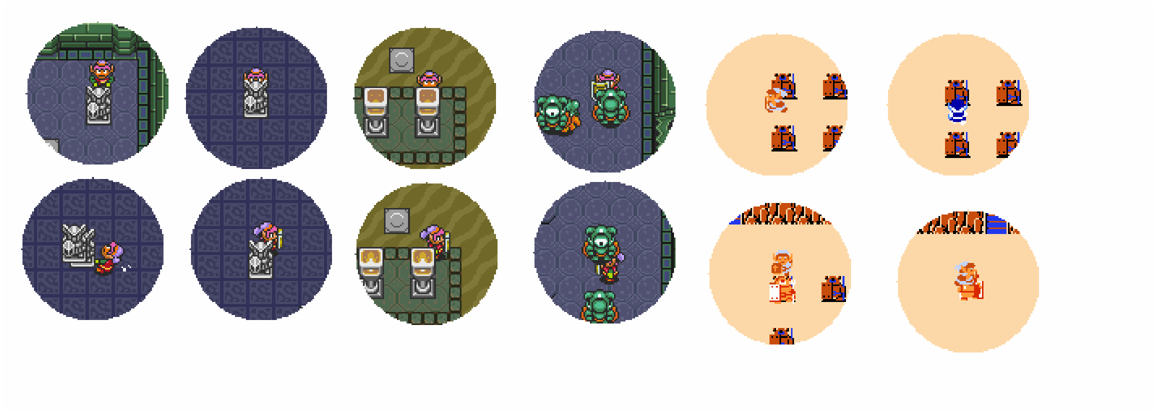
The latest development of physics simulation in games is the new soon to be released Physics Processing Unit (PPU) by AGEIA PhysX. There are many rigid body simulators out there, both open source and professional, but this is the first to take advantage of hardware. However the chip will mainly deal with physic effects that do not affect gameplay, such as hair animation, clothing, and anything that won't directly interact with your game world. Havok DX, the next version of the well known professional simulator, will use the Graphics Processing Unit (GPU) to get some of the same effects as the PPU. This means that there really isn't a best way or best engine to do the job for your collision tests; it requires creativity and ingenuity to choose the strategies that will provide the best interactions of objects for your specific game. The first thing many game designers do is create the collision and some animation code, because collisions have to relate to what the player sees, then create some test rooms, levels and objects to get the interaction perfect as possible before continuing with the game. Small quirks in an engine can ruin the whole game if players have to fight the game instead of focusing on the objectives in the game.



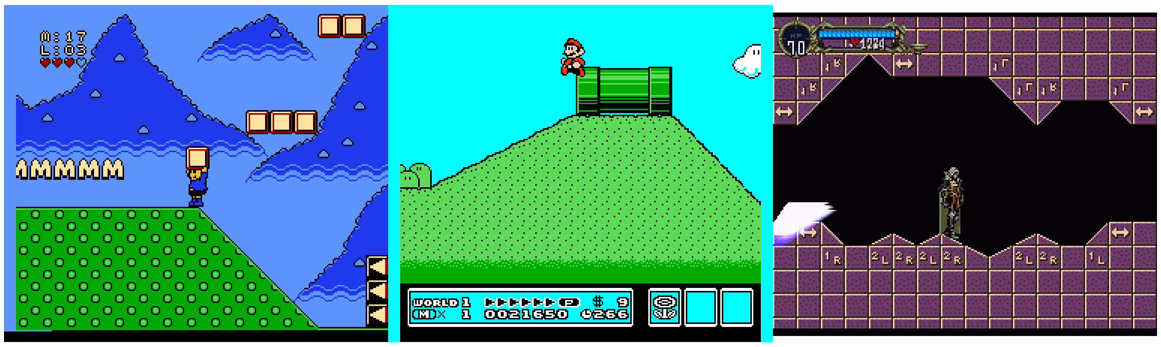
This illustration shows some techniques for enclosing a ship that doesn't fit onto an axis-aliened bounding box very well, and what happens when it rotates. Enclosing it in multiple AABB's gives you the best fit, but can only rotate multiples of 90 degrees unless you want to use boxes that aren't aligned. Circles are good for you objects If they rotate at any angle and you want to get away without writing functions to test collisions with oriented bounding boxes. Using the sprite bounds or a square that encloses the object at any rotation leaves a large amount of open space inside the box. They should only be used for the broad phase of your detection routine.



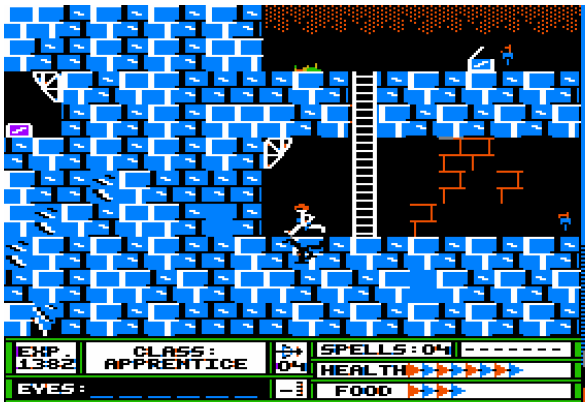
This illustration shows how you would go about setting up a hexagonal map. The first picture shows how you would define the coordinate system for the hexagons. The x index would increase running to the right, and the y index would zigzag down the map. There are many other ways to define the coordinate system, even a 3d version for finding points in a 2d hexagon, but this is one example. The next picture shows how you would split up the space into rectangles, which are much easier to determine which cell a point is in. Once you determine the cell, you use the bitmap (far right) to decide which hexagonal cell the point is in. The first thing you have to do is determine what cell the white space of your bitmap maps to for the rectangular cell you found yourself in. This is done by letting x index of the hexagonal cell equal the x index of the rectangular cell, then letting the y index of the hexagonal cell equal double the y index rectangular cell. Now if your point is in the white area, you know which hexagonal cell you are in. If you are in a different color region, you know where the cell is from the white one (the offsets of the x and y index for each region are labeled in the illustration).



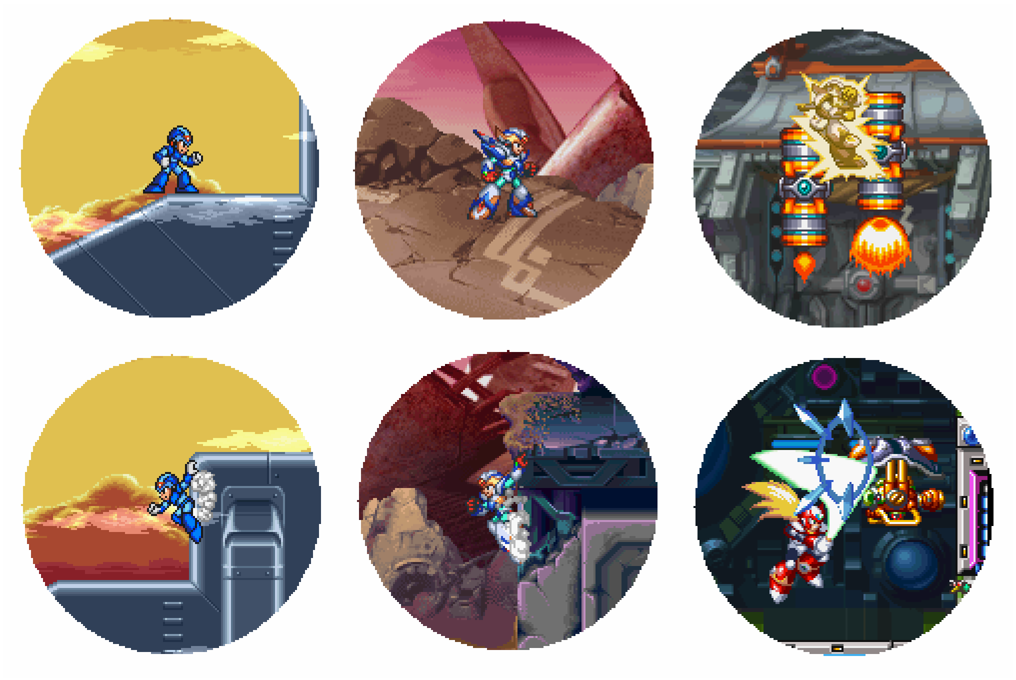
In these images you can see that the Legend of Zelda is a two dimensional game that's graphics were designed to give it a pseudo 3d look. The newer version uses smaller tiles than the original, which helps make the environment seem less "blocky". Some of the tiles behave as slanted tiles allowing Link to slide along them, and when he presses up against solid tiles when he is on the edge, they also allow him to slide so he doesn't get stuck on the map. Many clones fail to achieve this effect and that can greatly inhibit the enjoyment of the game. By looking at the upper left image, you can see that some objects collision detection is inconsistent with the system they were using. The second top and bottom image show Link behind a movable statue that is part of the object list. The third two show him behind a torch that is part of the tile map. But he can't pass behind the statue that is part of the tile map. It doesn't have a great impact on gameplay, but it could potentially create an exploit in the game. The object list isn't sorted by the y-axis to have objects rendered correctly, probably because they didn't have much computational power left. The only object they correct the priority of sprites is when Link is in front of a movable statue. When objects collide with other objects, a purely soft penetration constraint is enforced. Statues are objects that need to constrict the player's movement, so a hard constraint is used. (probably by pushing all other objects out of it) There is no logic the game runs for when two statue like objects overlap, as you can see from the lower left image. The original Nintendo Zelda game (to the right) had inconsistent collision methods for objects and map tiles. In one image you can see that Link is touching the statue graphically, but not considered to be colliding. The statues start out as part of the tile map, and when touched, they turn into objects that can hurt Link. The moment the statue turns into an object it damages Link even though the tile version was considered to not be colliding. Too many inconsistencies like this can confuse players and create frustration.



These are images of various games that use slanted tiles in their tile engines. They all use hotspots to detect collisions with them, but they all do it in a different way. Castlevania: Symphony of the Night (far right image) has a debug room that shows the tile types used to interact with the slanted shapes. The tiles that have double arrows are simply tiles that the player can pass though from the left or right. They need to be in-between the 45 degree angle tiles otherwise the player would never be able to climb all of the way up the slope before being stopped by the adjacent tile. Two of the types of slopes need to have identifier tiles under them because the hotspot that defines the player's position would be inside the identifier tile, not the slope when he walked by. This tile lets the system know that the player needs to start climbing the slope.

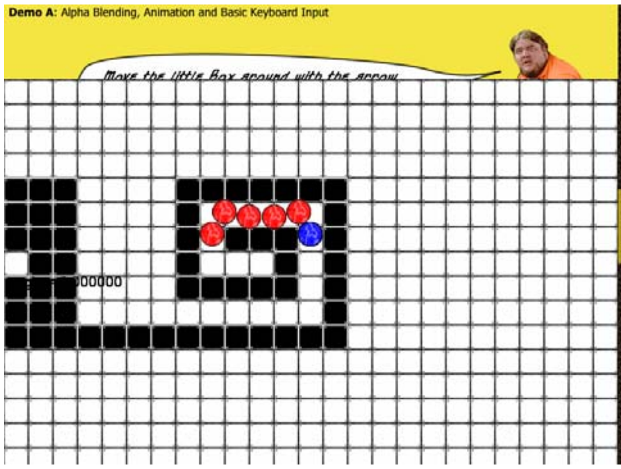
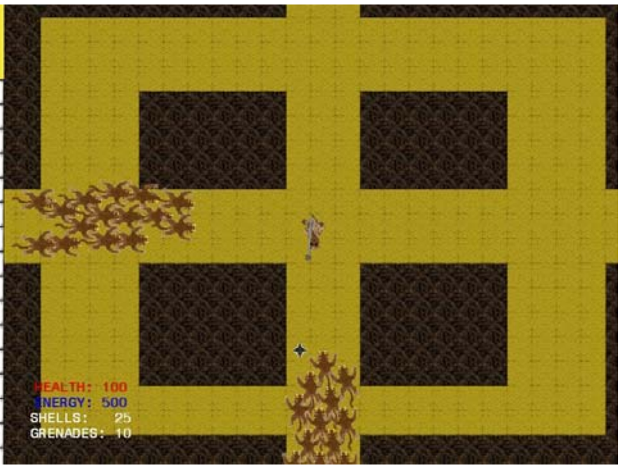


In the early days of video games people hadn't had a lot of experience designing collision detection systems, and a lot of games could be done by one or two people with some extra time on their hands. In the previous frame of the screen on the left, the player (the white red haired guy) ran into the wall, and the game pushes him way out of the wall, to the point that he can run quite a distance before he will run into it again. This isn't necessarily a problem with lack of processing power of the old platforms. Karateka did a good job of keeping the collisions smooth and consistent on the same platform (Apple IIe).

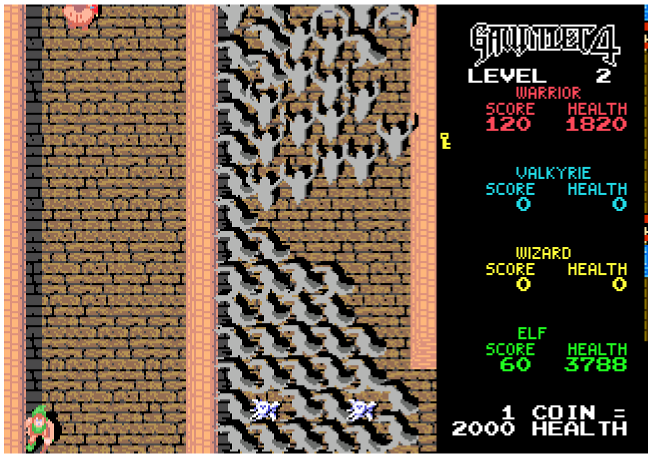


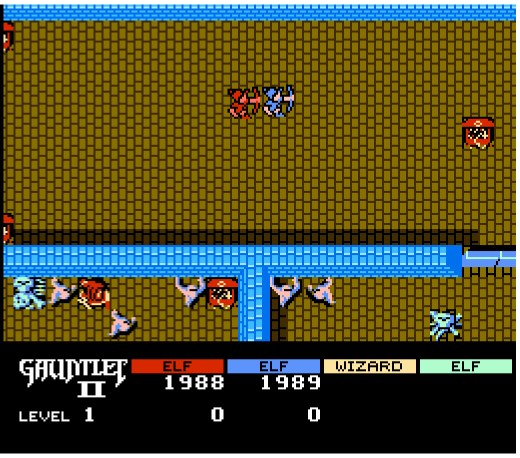
The Mega Man X series for Nintendo and later the Playstation had many interesting effects that can be difficult to reproduce. It had moving platforms that you could stand on or slide down and they would crush you if you got wedged in between them (upper right image). An algorithm that simply ran though collisions detected wouldn't be sufficient because you would have to know about the other moving platform positions before you could determine how the player should react to a collision with a single moving platform. The game also had entire tile grids that would crush the player or move away exposing a pit the player could fall down. There were also ways to wedge objects in between the grids to prevent them from closing all of the way. Some enemies had shields that would prevent bullets from hurting them (lower right image). This was either achieved by determining which side the bullet entered the enemy, or using two separate bounding boxes for the shield and vulnerable part of the enemy. Ray casts would be useful here so the game would know which box was hit first, otherwise bullets may get though. This would especially be true for Zero's sword that is very long and always goes though the vulnerable part of the enemy. As the series and hardware evolved, the graphics were improved, but the collision schemes seemed to remain the same. The only change was adding tiles above the ground that looked like they extruded into the background, and making the players feet look like they were at different depths.

Without the tiles above the ground, it would look like the player's other foot was hovering in air. The original versions just make his feet rest on the same plane because the game had much less of a pseudo 3d look. In the upper middle picture, the player is standing on a slope that appears to be 3d.

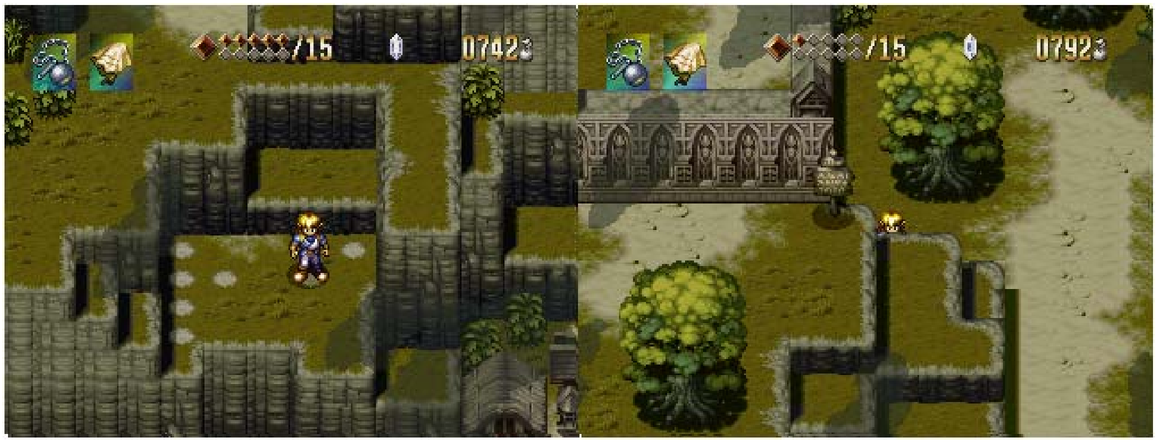

Demo A: Alpha Blending, Animation and Basic Keyboard Input
Move the little Box around with the arrow
000000


This is a game I did for a 48 hour contest. To the left is an early shot of the game. The first thing I did was make place holder graphics and a test level to figure out what my collision detection system could handle without creating too many anomalies. In that picture, you can see that the blue ball is pushing all of the red balls though a small winding hallway. In order to get this effect I had to sacrifice some accuracy for functionality. All this algorithm does is run though all of the objects, and for every pair of objects that overlap they are moved out of each other using a least penetration vector. All of objects and tiles are circles, so calculating the least penetration vector was simple. The list of objects is sorted along the x-axis so the results are dependent on their position along the x-axis and whatever order objects with equal x coordinate values happen to be sorted in. If an object collides with a tile, the object is moved out of the tile. If two objects collide, they are moved along half of the least penetration vector in opposite directions. There was going to be a third kind of object that moved that would move objects out of it, but wouldn't collide with the tiles. This could be used for doors and objects that monkeys or players shouldn't be able to push around. The grenade doesn't bounce off walls the way it should because the tiles are actually circles, but it could be tweaked to approximate which side of the tile the circle collided with.

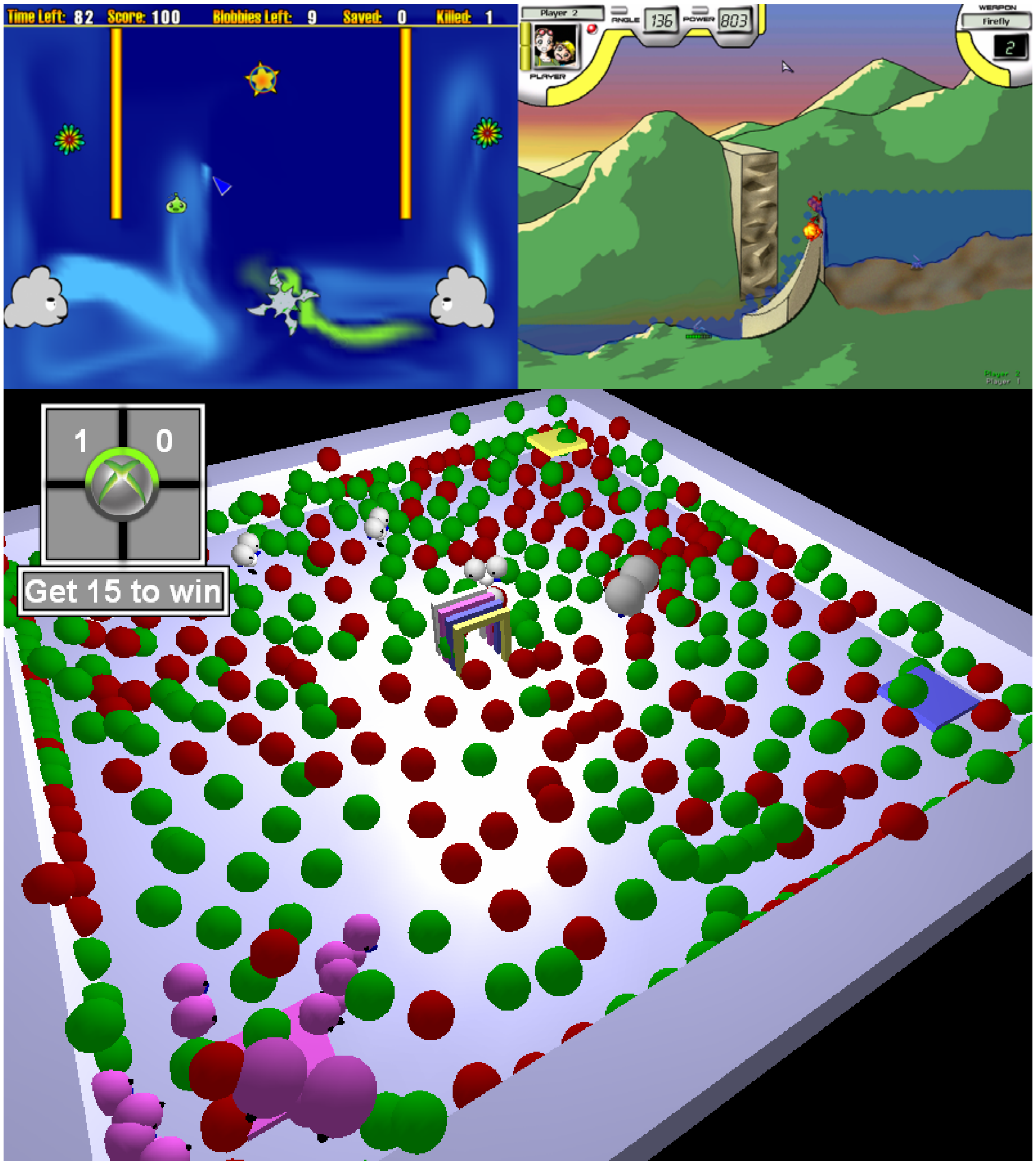




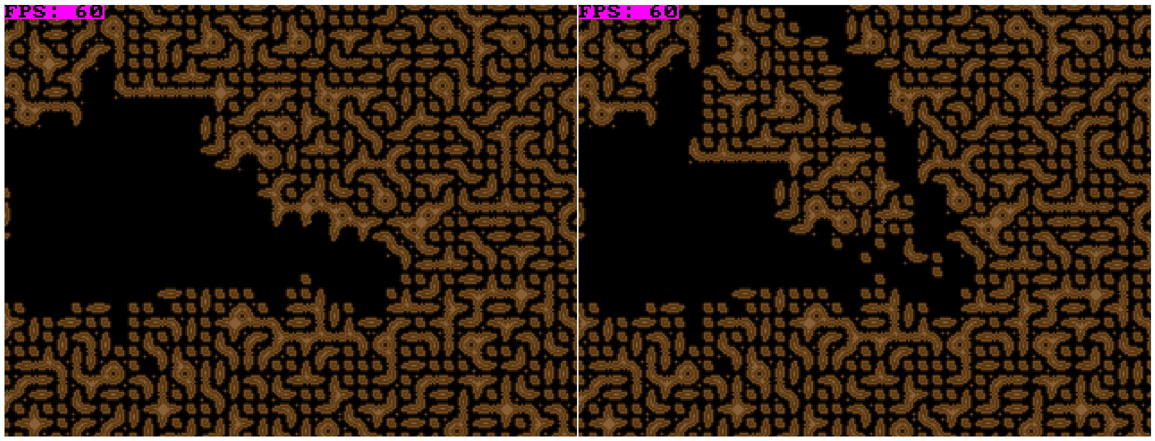
Gauntlet solves the penetration constraints of its objects very well. There are no jitter effects even when there is a large room of monsters, and every object in the game is active even when off the screen. The game doesn't allow players or monsters to push other monsters around, but it does have routines that allow players to push each other around. The monkey game lets anything push anything else around, but when there are too many objects, there is tons of jitter, to the point that monkeys will fly though walls. Some of the ports of Gauntlet are made well without jitter, others have some strange side effects in different situations. Try to have two players push each other around in the Nintendo version of Gauntlet II, then try it in the Genesis version of Gauntlet IV. There is a distinct improvement in the handling of penetration constraints in the latter game. There is a flash version of gauntlet online that simply avoids solving any penetration constraint all together, and allows objects to go wherever they want.



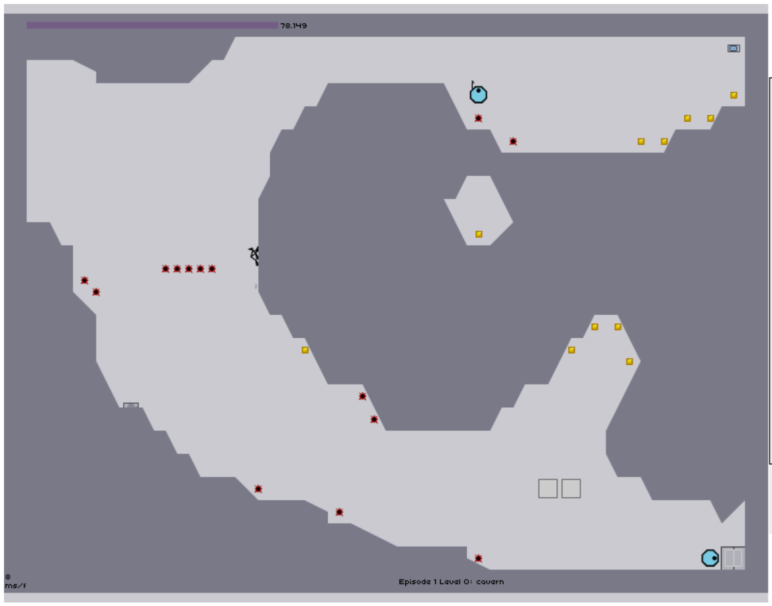
Alundra was a Zelda style game for the Playstation that used a true 3d tile system. The player could walk behind tall cliffs and ride rising platforms to gain access to higher floors without using any visual tricks. This game could probably be ported to 3d without changing the collision detection system at all and still look right. When using a 2d fixed camera, level designers had to make sure that the play area wasn't covered by tall walls and buildings in the foreground.

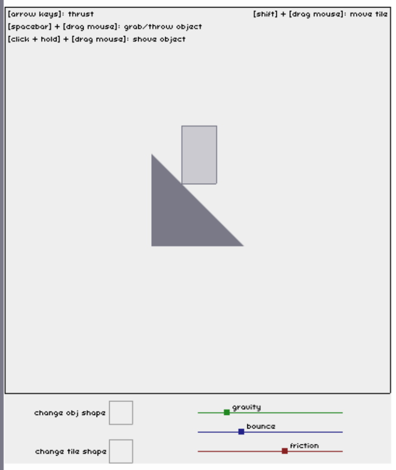


Blobby Waters (upper left) uses a grid to store a vector field that determines what forces the current will apply to the blobbies to drag them around. Every cell has to be updated every frame so the grid needs to stay relatively small. Adding and removing solid tiles to the grid can be done on the fly and have immediate effects on the current. Charred Dirt (upper right) uses a simple particle system to get fluid effects in their game. It isn't very realistic but it has general properties of liquid, which is sometimes all you need. Play Place (bottom) has a more sophisticated system uses Navier-Stokes equations and a kernel to determine what other particles each particle is directly affected by. It seems to sacrifice quantity for quality, which isn't always what you want. Penetration constraints between particles is totally done with soft constraints.

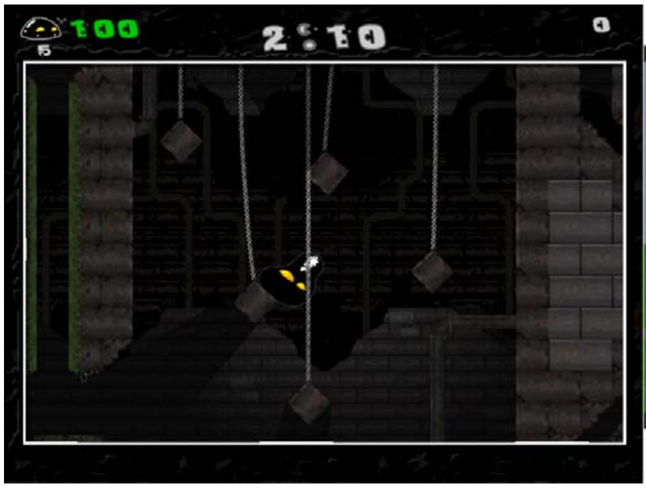


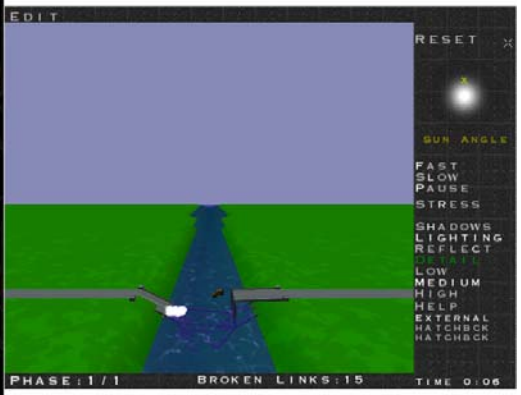
This was an engine that was going to be used with the Blobby Waters current engine to create a wizard game where you got to control the elements. There were also plans to use a simple particle based liquid system. Using too many of these together may have brought execution to a crawl. This system also used slanted tiles which may have been incompatible with the current system, unless we simply let wind current pass through them. As tiles are destroyed, it checks tiles above to see if they are still being held up. Once they fall, they are put into an object list until they hit the ground and settle into the cell they fell into. This allows the engine to ignore any tiles that are in the grid and only apply gravity forces to ones that are in the list. I was able to create huge map sizes with little to no slow down even with the constant random removal of objects.



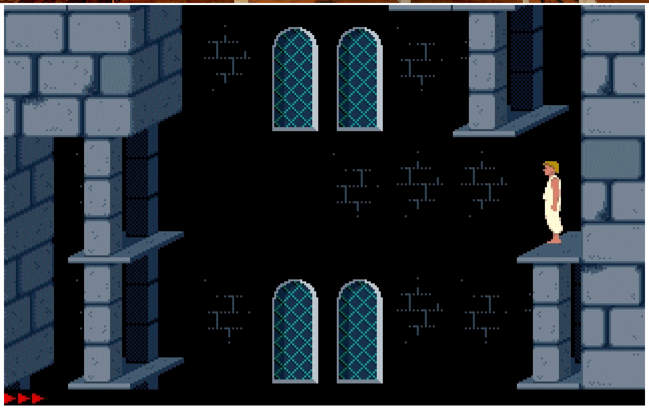

[arrow keys]: thrust
[shift] + [drag mouse]: move tile
[spacebar] + [drag mouse]: grab/throw object
[click + hold] + [drag mouse]: shove object

The flash game "N" was created to demonstrate decent collision detection to a community of flash game designers whose detection schemes were poor at best. It uses a combination of tile maps, ray casts, object lists and an accurate narrow phase collision detection routine for colliding with various types of shapes. It uses a principle called the Separating Axis Theorem to test for collisions of various shapes, and can be used to calculate the vector of least penetration between two different types of shapes. This game allows the sorting of the game object list to affect how collisions are resolved and there is potential for objects to be moved back into each other from enforcing penetration constraints with other objects. However it is difficult to tell even when many objects are colliding at once. Using boxes for characters may look strange for this engine because once on slopes, it would look like they were standing on one toe. You could change the graphic to make it look like they had one foot on different parts of the slope, but then the lower leg wouldn't collide with anything. Most tile engines use hotspots instead to make characters look like they are standing on slopes by making only the center of the box collide with the slope.

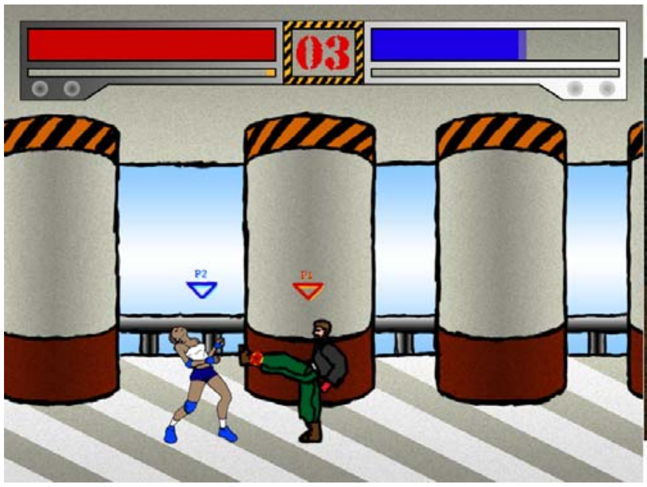


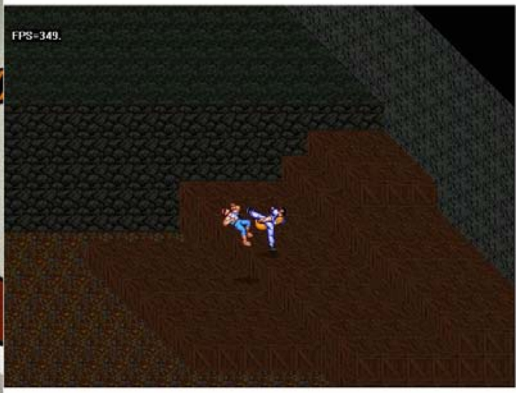


Gish proves that we haven't come close to running out of concepts for 2d games yet. It uses a familiar tile system with slanted tiles along with a rigid body simulator for objects that interact with the tile map. It even has a good rope and chain simulator that allow them to interact with the tile map as well. Gish behaves almost like a liquid that can alter its heaviness and friction with the environment. I have never seen another game simulate a blob so well. Chronic Logic, the company that made Gish, also made many bridge simulation games. Their games show that physics simulators defiantly have their place in games to enrich the game play.

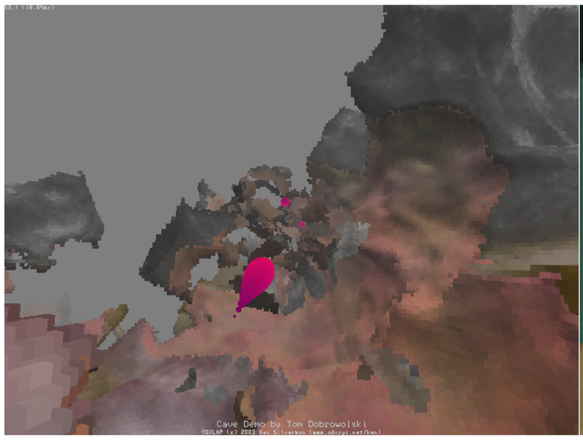


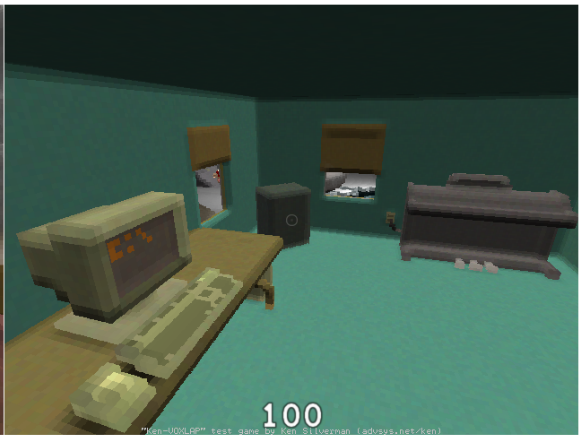
Tomb Raider was one of the first 3d games to capture the game play of a hit game called Prince of Persia. Thank god the success of this game didn't rely entirely on the main character's breasts. Prince of Persia had a unique environment that allowed the character to figure out what actions that they should do based on context. For example the character knew that he should wait for the last minute to jump over large pits without the player having to do the exact timing. It also prevented the player from stepping on dangerous switches and spikes. Tomb Raider used a 3d tile system so it could capture some of the same features that Prince of Persia had, such as smart jumping and climbing.



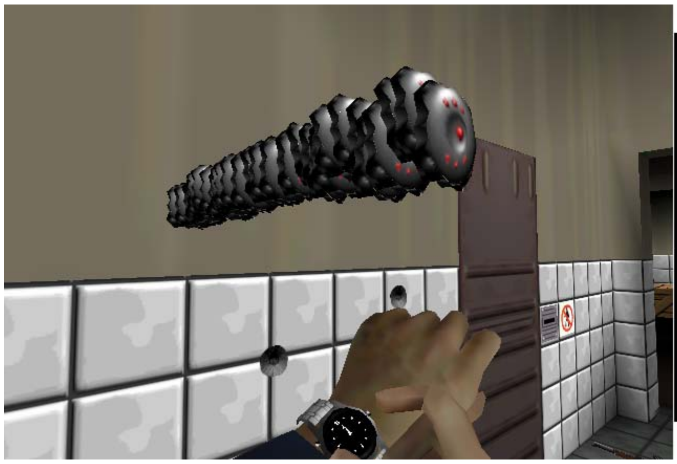


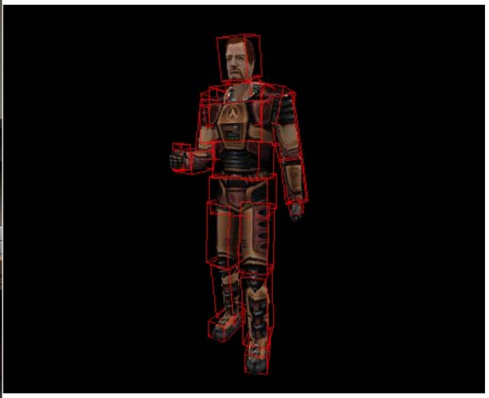
To the left is a game called Red Reflex, a game that uses a different set of hit boxes for every frame of character animation. They were placed with an editor to speed up the development process. The only penetration constraint used is the hotspot that defines the characters position with the ground. Every frame had to be offset to make it appear that the hotspot was the first point visually that would hit the ground. The penetration constraints used to prevent characters from passing though each other were taken out at the last minute because the characters would jitter around which created several anomalies that would confuse the player. To the right is a fighter demo that uses planes to define the environment, and hit boxes the same way Red Reflex used them. It would determine which way objects should be moved out of walls by their velocity vector and their last position, and even enforced penetration constraints with every hit box on the character. This posed a huge problem when I realized that hit boxes wouldn’t have and velocity or last position because they would appear as new frames were animated. If the player was standing still, the boxes could change position so instead I created a constant bounding box that collided with the environment, and let the limbs of kicks and punches go though walls. This game also doesn’t render correctly, if a player goes behind a plane they are still rendered on top of it. This engine needs to sort objects along with the planes, probably similar to how a BSP Tree does it.



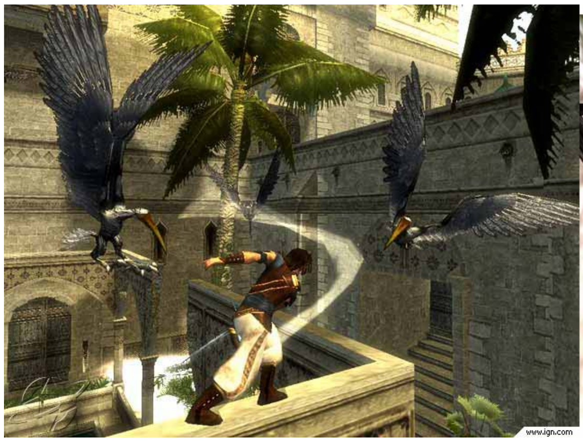


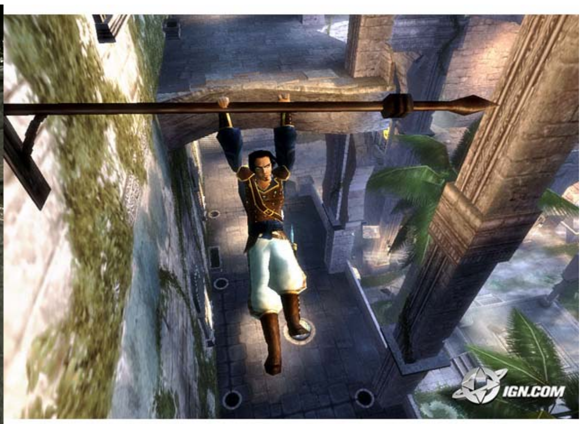
Worms followed in the footsteps of a popular game called Scorched Earth and used a pixel based tile map that allows for easy destruction of landscapes. When the game was ported to 3d, they had to scrap the pixel based method because the industry is obsessed with polygons to make the games graphics good. Instead levels were made up of larger objects that disappeared when hit with some sort of projectile. Many people have tried to port Scorched Earth to 3d by using height maps, but those games feel more like another 2d version from above instead of the side. Someone has yet to make a true pixel based Scorched Earth game (pixels are referred to as voxels in 3d). Fortunately a man named Ken Silverman develops voxel engines for games after making the build engine for Duke Nuke'em 3d, Redneck Rampage, Blood, and Shadow Warrior at the age of seventeen. When you have those kinds of royalties under your belt, you can work on whatever you please. Another noteworthy feature of Ken's engine is how particles that are suspended in air fall to the ground. A kernel function is used to determine when particles are no longer attached to the rest of the world. You can tell how big it is by attempting to cut a hole all of the way around the house in the demo which should make it fall, but stays suspended in air. Since there are no polygons, its difficult to determine surface normals. So what the engine does is estimate the normal at a give point using the voxels around that area.





Half-life uses a series of oriented bounding boxes to enclose their characters. They are only used to determine location damage for bullets, they are allowed to pass though solid objects. A simpler shape is used to make sure the characters don't walk though walls. This can have a serious impact on gameplay if walls are too thin. One game I played I was able to kill someone who's knee was sticking slightly outside of a thin wall to get an easy kill. Quake 3 uses simple shapes to enclose their characters with no location damage. This makes it easy for the modelers who don't have to pay attention to the geometry of the gameplay, for the game generates the bounding box from the model. This became a problem for smaller characters that had a distinct advantage in the game. For tournament games these smaller models weren't allowed to be used. When you are playing online first person shooters, most of the time what you see is only a late approximation of where the objects are in the game, so high precision aiming is difficult to have any sort of consistency. That is the reasoning behind removing the location damage used in the previous tiles with a similar engine. Goldeneye 007 and Perfect Dark use a similar system as Half-life, the main difference being that once a location area was determined, they took the extra step to calculate the exact polygon that was hit by a bullet. This made some interesting events happen (such as a grenade you just threw getting hit with a bullet exploding in your face). Perfect Dark added the event of characters dropping their weapon if it was hit by a bullet. The game still had the same problem Half-life did, limbs sticking out of doors could still get shot.





Prince of Persia: The Sands of Time had very smooth player interaction with the environment. Instead of coming up with algorithms that would allow the player to interact with any arbitrary set of polygons, they first decided how the player would interact with the environment and created engines to test their strategies. They perfected their game objects before any final level design was completed. This way they could allow more complex interaction with objects, and allow the engine to support fighting and exploring at the same time. In the left image the player is doing combat on a balance beam that gives you special controls to navigate, and allows you to whip out your sword, jump, attack, all without any quirky behavior that makes him fall. And when he does fall he automatically grabs the edge. In the right image the player has grabbed onto a pole that he can swing from and even gain momentum to jump off farther. The game also has beams that can be grabbed on to and climbed. It is a lot easier to define objects and place them in your level than to try and determine how a player should interact with an arbitrary set of polygons, i.e. tiring to determine whether an object could be climbed or grabbed like a pole. A similar method is used in skating games where you can do tricks off of any object.

These are two different images from fighting/adventure games. The both use sophisticated fighting and exploration engines. Die by the Sword (right image) uses a whole kinematics system to animate and control the character. An analog device is used to move the weapons arbitrarily though space. Penetration constraints are used on every limb and weapon that is being carried with the environment. It even calculates how hard you hit your opponent, which can be increased by coordinating your swing, step and turn movements. It’s really important to have constraints solved correctly here, because thin swords need to be used to block other thin swords (which could pass though each other if the time step is too large). Because it knows where limbs are hit, they are severed accordingly. Although this was an ambitious title, some anomalies were difficult to track due do the complexity, and coordinating body movements on keyboard and mouse take longer to learn that learning the real life body movements to swing a sword. Blade of Darkness (left image) uses more of a traditional fighting game approach by allowing the character to be controlled by actions instead of arbitrary movements. The only real simulation done in the game is with inanimate objects and thrown objects. Limbs can also be severed, but it is more of an effect than a game play feature. It only detects general areas where bodies are hit. Weapons and limbs aren't constrained by walls, but do make a sound when they collide with them. This game was much easier to pick up and was more consistent physically than Die by the Sword, even though it was more "realistic". Another game that uses an interesting collision detection scheme for a fighting engine is a Playstation game called Bushido Blade. When you weapons collide with walls, the player animation is simply reversed to make it look like the weapon bounces off the wall.

References & Resources

Freeware Games:

Zelda Clones:
http://www.graalonline.com/index.php?topic=classic
http://www.vx4.com/strategy-games/zelda-game.html

Megaman Clones:
http://www.stroutsink.com/mm21xx
http://www.t45ol.com/play_us/850/megaman-project-x.html
http://www.2flashgames.com/f/f-775.htm

Castlevania Clones:
http://freewebarcade.com/games/castlevaniabloodway/castlevaniabloodway.php

Gauntlet Clones:
http://snowstorm.sourceforge.net/cgi-bin/site.cgi
http://www.gamesforge.com/showflash.php?file=gauntlet[1].swf&album=1

Flash Game “N”:
http://www.harveycartel.org/metanet/n.html

References:

Physic Simulation Papers:
http://www.d6.com/users/checker/dynamics.htm

Constrained and Unconstrained Rigid Body Dynamics:
http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/user/baraff/www/pbm/rigid1.pdf
http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/user/baraff/www/pbm/rigid2.pdf
http://www.cs.rutgers.edu/~dpai/papers/ClinePai03.pdf

Siggraph:
http://www.cs.cmu.edu/~baraff/sigcourse/index.html

Fluid Mechanics:
http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf
http://cg.informatik.uni-freiburg.de/course_notes/ca_01_gissler_sph.pdf

Havok FX Article:
http://www.xbitlabs.com/news/multimedia/display/20051028224421.html

Programming of MC Kids:
http://www.greggman.com/mckids/programming_mc_kids.htm

Ubi Soft interview:
http://www.cgno.com/features/29.html

Ageia Homepage:
http://www.ageia.com/developers/support.html

Ken Silverman Voxel Engine:
http://www.advsys.net/ken/

More Articles:
http://www.gamedev.net/reference/list.asp?categoryid=45#199
https://www.gamasutra.com
http://www.devmaster.net/articles/quake3collision/

Screenshots From:
http://www.ign.com
http://www.gamespot.com