JK ENGINE RESEARCH

2018 BAD ASS HACKERS

TABLE OF CONTENTS

INTRODUCTION

This document attempts to demystify the effect of various behaviors in JK.

TOOLS

For testing, a new level ("smithdev") has been created which contains several test chambers and COG scripts to aid in collecting information for the reverse engineering efforts.

A toolkit called jkinjector.exe is used to dynamically register custom COG syscalls to JK, which the "smithdev" level scripts are able to use to log data. Particularly in the case of physics, an additional tool.exe can be used to format this data into a FreeMat file for subsequent plotting.

It is these toolkits which will be used in the findings of this research document.

“smithdev” level

This section describes the areas used in the level.

Thrust Chamber

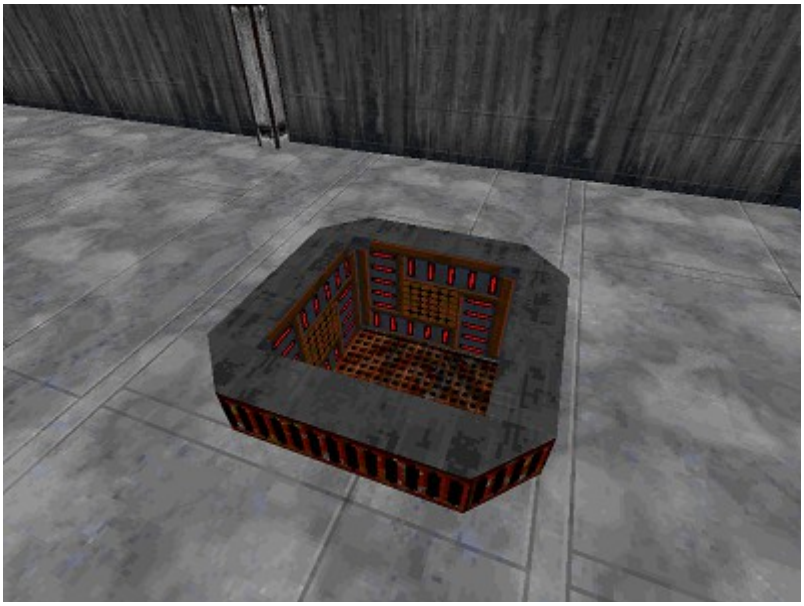


Illustration 1: "smithdev" Thrust Chamber

This chamber will output the player's thrust vector to the screen. It is designed so that the player will remain stationary during use.

It is convenient to change the bottom surface's flags to test how different surface types may affect the player's thrust.

Thing Flags Chamber

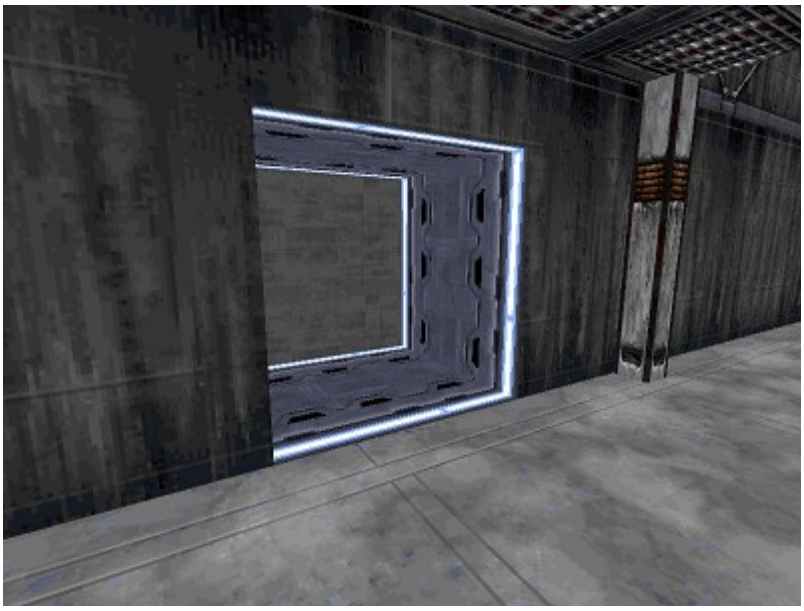


Illustration 2: "smithdev" Thing Flags Chamber

This chamber will output the thing, physics and typeflags of any thing that crosses the adjoin into it.

The script uses a mask of -1 to allow detection of any thing type.

It can be used to check these flags of the player or projectiles, for example.

Velocity Hallway

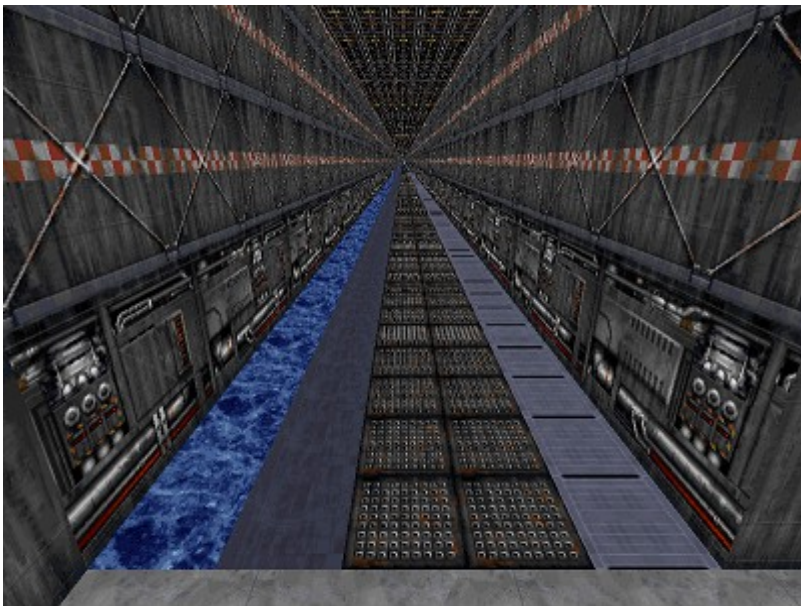


Illustration 3: "smithdev" Velocity Hallway

This extremely long hallway will output the timestamp, thrust and velocity information to the screen on a continuous basis while the player is in it. Also, if jkinjector toolkit is running, the custom “jacklogvel” COG syscall is used to capture this data to file.

The hallway is long enough to allow for the player to reach peak velocity while running, or to use other effects such as Force Speed, without running into a wall.

The different strips of floor pattern are used to test different surface flag types. For example, the dark blue strip (second from left) has no Floor flag so it can be used to check how being unattached to a floor surface affects which drag coefficient(s) are used.

Actor Control Chamber

Illustration 4: "smithdev" actor control chamber

This area contains an AI character and switch(es) to initiate various tests.

reveng

This section describes components of the reverse engineering toolkit.

jkinjector

This application provides the mechanism for DLL injection and basic console output.

When executed, the program will continuously scan for an instance of JK. Upon finding a match, an exploit using the kernel32 API is used to force the target to load the "jkjacker" module into its process space.

jkjacker

This DLL is the payload of jkinjector and provides several features including registration of new COG syscalls and procedurally-generated code for handling their parameters and return values.

tool

A very simple application for parsing raw data dumps and outputting FreeMat (*.m) files for subsequent plotting. In particular, this utility is used for plotting velocity graphs over time.

PHYSICS

Integration Schemes

In this section we will explore some concepts of integration: the process of updating velocity from acceleration and updating position from velocity. Generally, this is done once per object, once per game engine frame.

The full details of integration schemes is out of the scope of this document, however some common types will be mentioned.

This section is here to serve not only as a refresher, but also because some of the analysis requires selection of an integration scheme in which to derive, for example, acceleration forces from measured velocity changes. **Further analysis sections will utilize Euler for its simplicity; this may result in some anomalies if JK or Smith utilize a different integration scheme.**

The delta-time (*dt*) represents the time, in seconds (or moreover, small portion of a second) between the current engine frame and the previous engine frame.

During collision detection, it is common to use the previous-current dt to predict the current-next dt so that collisions can be predicted; rather than letting objects actually penetrate each other. The manner in which collision detection/resolution is performed may, in large part, dictate the integration scheme most desirable.

Euler

In the Euler scheme, an object stores position and velocity as state variables.

```
struct Object
{
	Vector3 Position;
	Vector3 Velocity;
};
```

The object's velocity is updated by its current forces (acceleration) using dt. Then, the object's position is updated by its current velocity using dt.

```
Velocity += Acceleration * dt;
Position += Velocity * dt;
```

It is therefore possible to derive an unknown value, such as acceleration, by examining velocity across the span of two engine frames. **This will be the method in which this document uses.**

```
Acceleration = (Velocity – LastVelocity) / dt;
```

There is some chance that JK may use this scheme, due to the existence of COG syscalls such as "SetThingVel" which appear to be able to modify an actual velocity state variable.

Verlet

In this scheme, an object forgoes storing its velocity as a state variable and instead stores the current and previous positions.

```
struct Object
{
	Vector3 Position;
	Vector3 LastPosition;
}
```

These two values can be used to determine the velocity on the fly.

```
Velocity = (Position – LastPosition) / dt;
```

Conversely, it is possible to “set” an object's velocity by artificially changing it's last position.

```
LastPosition = Position – Velocity * dt;
```

If JK uses this integration scheme, then this “recalculating last position” may be how a COG syscall such as “SetThingVel” would work.

Velocity-Verlet

This method is an extension of Verlet which allows storage of velocity as a state variable.

Further details are not discussed in this document at this time.

Runge-Kutta (RK4)

This method is similar to Euler or Velocity-Verlet in that an object's position and velocity are stored as state variables. Generally speaking, it is touted to be more accurate since it essentially layers several stages of integrations.

Further details are not discussed in this document at this time.

ANALYSIS

This section provides samples of collected data and initial analysis of results.

PHYSICS

Thrust

Thrust may be applied by several methods, and it is yet to be determined the conclusive list of possible sources.

It is known, for example, that player movement relies on thrust.

However, the gravity force appears to be applied directly as an acceleration, bypassing the thrust arithmetic. This is evidenced by the fact that falling downwards [by gravity] does not appear to affect the thing's thrust value.

Applying Thrust

It is not yet confirmed how thrust ultimately is applied to update a thing's velocity.

Considering that player movement thrust is always done upon the Y axis for forwards/backwards and X axis for strafe left/right, it is assumed that thrust applies a force in the look direction of the thing.

An initial implementation in Smith engine effectively calls ApplyForce with the thrust vector transformed by the look direction of the player, and this appears to have the desired effect.

Of further interest is AI actor movement also appears to be thrust-based (described later), yet an actor will tend to begin moving towards its target location even when still turning to face that direction.

Player Movement

The initial analysis was performed prior to the creation of this document. Therefore, not much raw data will be presented here.

The following figure originally contained raw readings from the Thrust Chamber under various circumstances, but this has since been consolidated.

NOTE: Looks like template property 'maxthrust' is used to determine baseline forward walk speed. Eg., walkplayer defines maxthrust=2.0
NOTE: If using math to calculate directional thrusts by single number (eg. Forward is 2x Backward speed) then ActorExtraSpeed still needs to be added AFTER. It is a static increase regardless of direction.

Movement	Axis	Walk (surface)	Calculated
Forward	Y	2	maxthrust
Backward	Y	-1	maxthrust*0.5
Strafe Left	X	-1.4	maxthrust*0.7
Strafe Right	X	1.4	maxthrust*0.7

SetActorExtraSpeed	Simply adds value (BEFORE any other mod)

If not attached to surface, final thrust is cut to 30% (crouch/run/surfaceflag mods are ignored, but extraactorspeed was already added)

Deep Water 0x20000	50% normal surface
Shallow Water 0x40000	100% normal surface
Very Deep Water 0x100000	50% normal surface

NOTE: Icy (0x1000) and Very Icy (0x2000) do not have any effect on thrust. Most likely they just cut surface drag by some percentage.

Run (ignore if crouching!)	Double thrust
Crouching	Cut final thrust by additional 50%

Text 1: Actor movement thrust rules

Note that although this research, so far, is performed only on the Player (walkplayer), it is suspected that other Actor types utilize the same thrust logic. A thrust monitoring script on an AI actor should be performed in order to confirm the theory.

The templates for various AI actors redefine maxthrust to a value that seems consistent with the move speed of the actor. For example, the gonk droid (which is known to move rather slowly) defines a maxthrust of 0.2 which is considerably lower than sentry droid (which is able to move at a decent speed) whose maxthrust is 0.8. Therefore, it is suspected that the movement of AI actors is also done via application of thrust according to the above table.

Actor Movement

One notable effect when monitoring thrust is that player thrust most definitely appears when calling “GetThingThrust” from COG [even if the player is in such a position that doesn't allow movement]. Contrarily, calling “GetThingThrust” on an AI character seems to report a 0 vector even when the actor is most definitely moving. Note that a test was also performed using a robotsentry, a flying actor, which also produced the same result.

A velocity analysis on actor movement shows a very similar result to the player's movement.

Further, the maxthrust template property of actors seems to correlate with their perceived movement speed. Indeed, creating a new actor template with a modified maxthrust property does result in the actor moving at a corroborating movement speed.

Finally, the physics flags of actors tend to include the 0x2 “uses thrust to move” flag.

Therefore, it is assumed that actors do move using thrust. Perhaps the reason why it does not appear in COG “GetThingThrust” is because either:

(a) the thrust vector is applied directly as a force/acceleration, eg. “ApplyForce”.

or

(b) the point at which COG VMs are processed is after a point in which actor thrust has been applied and reset to 0, whereas player thrust may be handled at a different point.

Noted previously, an actor will begin moving toward its indicated move target even before fully facing that direction. This would correlate with condition (a) whereby a thrust is applied directly, rather than in the look orientation of the thing. There may as well be additional logic by other AI instincts such as Dodge and CircleStrafe that rely on being able to apply a [thrust-based] movement to the actor without having to change its look direction.

One final note regarding actor movement, if based upon thrust that is independent of look direction: the player movement penalties on “maxthrust” due to strafing, and especially moving backwards, may not apply to actors. This is purely conjecture and requires confirmation, but in recollections of playing the game, the author has no distinct memory of an enemy moving backwards any slower than moving forwards.

Drag Coefficients

Thing templates define three different types of drag coefficients: surfdrag, airdrag and staticdrag.

The default drag coefficients for the player (walkplayer) are defined as follows:

staticdrag	0.3
surfdrag	3
airdrag	0.5

Table 1: walkplayer drag coefficients

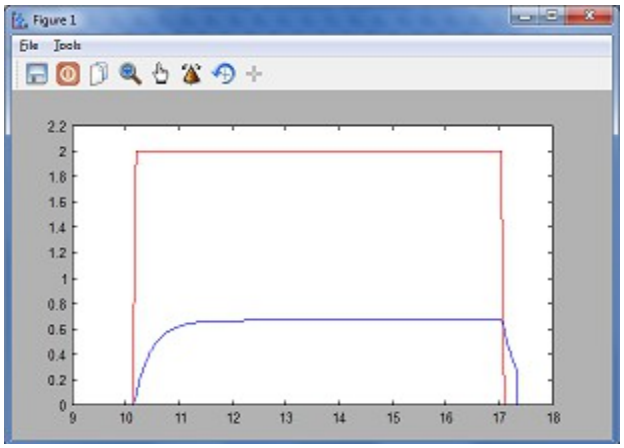
It is thought that not much has been published on the effects of these values or when they take effect, however it seems obvious that surfdrag is used as friction against a surface and airdrag is used as air resistance when not against a surface. It has been thought that staticdrag may be a fixed baseline that is always in effect, however the findings in this document seem to indicate this is not the case.

Some of the analysis methods provided here for investigating drag coefficients have some overlap to those used for investigating velocity and other forces.

Baseline

To begin, let's look at an initial plot produced by using jkinjector.exe to collect data from the Velocity Hallway area of the "smithdev" level, converted using tool.exe, and finally plotted using FreeMat.

This test was done by simply walking on a flat surface that is flagged as floor.



Drawing 1: Thrust/Velocity plot playervel_walk_normalsurface.m

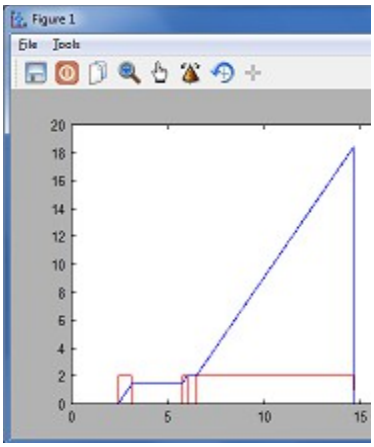
staticdrag	0.3
surfdrag	3
airdrag	0.5

Table 2: walkplayer drag coefficients

The red line shows the magnitude of thrust over time, and the blue line shows magnitude of velocity over time. Note at time 10 seconds when the player begins walking forward, the thrust magnitude jumps to 2. At this point, the velocity begins to increase in a non-linear fashion until it peaks at roughly 0.65. At 17 seconds, the player stops trying to walk forward and therefore the red thrust line drops to 0. Also at this point, the velocity begins to decrease in a seemingly linear fashion until roughly 0.3 magnitude in which case it drops to 0 seemingly instantly.

No drag

For comparison, let's look at a plot where the surfdrag, airdrag and staticdrag coefficients of the walkplayer have all been set to 0.



Drawing 2: Thrust/Velocity plot nodrag.m

staticdrag	0
surfdrag	0
airdrag	0

Table 3: walkplayer drag coefficients

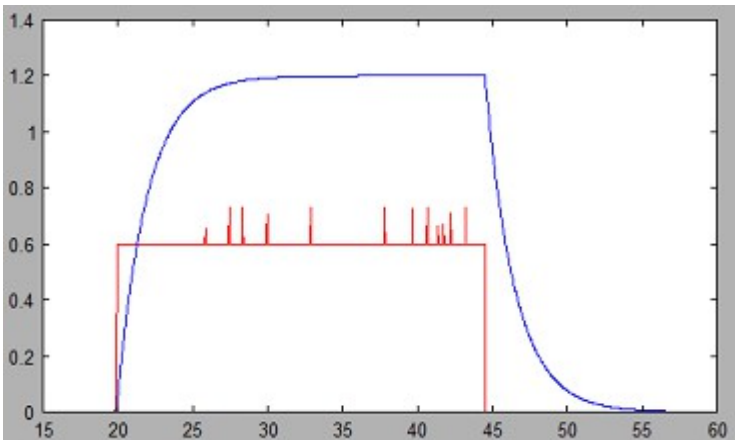
At roughly 2.5 seconds, the player begins to move forward; we can see what appears like a perfectly linear increase in velocity over this period. At roughly 3 seconds, the thrust (red) drops to 0, however the velocity is maintained exactly. At roughly 6 seconds, forward thrust is resumed and we can watch as the velocity appears to increase linearly to an extremely high rate of speed. **It is assumed that there is effectively no limit to the velocity that can be achieved when drag coefficients are 0**. At roughly 15 seconds, the velocity drops to 0 because the player has collided with the end of the Velocity Hallway and died from impact damage.

Next we will work toward isolating the rules of when the different drag coefficients are used.

For the following examples, we will return to using the default walkplayer coefficients as described previously.

Non-floor surface

In this next sample, we see the player attempting to walk on a surface that is not flagged as floor.



Drawing 3: Thrust/Velocity plot playervel_walk_nonfloor.m

staticdrag	0.3
surfdrag	3
airdrag	0.5

Table 4: walkplayer drag coefficients

Please note that the small thrust spikes (red) are slight strafe left/right taps to prevent sliding off the surface.

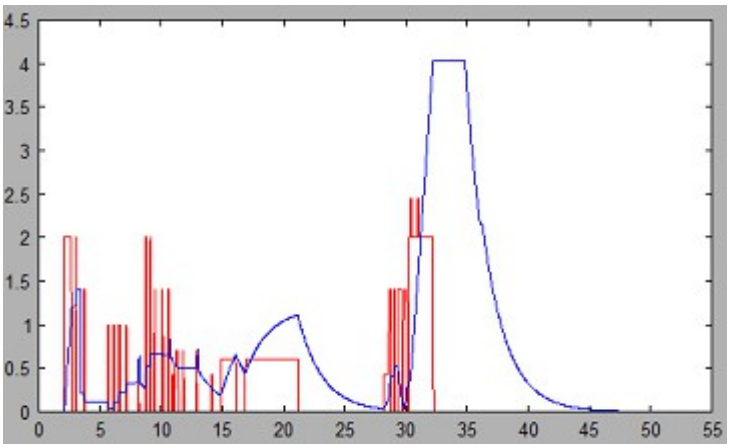
Overall, we can see that the thrust magnitude is 0.6 which is consistent with the movement thrust rules of cutting to 30% when not attached to a surface:

walk thrust	non surface thrust cut	final thrust
2	0.3	0.6

If we compare to the original plot of player walk on normal surface, we find that the player's peak velocity at thrust 2 was approximately 0.65. In contrast, in this test of player walk on non-floor surface, we find that the player's peak velocity at thrust 0.6 was much higher at about 1.2. This suggests that the drag coefficient on a non-floor surface is much lower than on a floor surface.

Airdrag

Next, we'll adjust our player's drag coefficients so we are only looking at airdrag.



Drawing 4: Thrust/Velocity plot onlyairdrag.m

staticdrag	0
surfdrag	0
airdrag	0.5

Table 5: walkplayer drag coefficients

There is a lot going on in this plot so let's take a moment to explain the two main sections.

In the period of 15seconds to 30seconds, the player is walking forward on a non-floor surface.

We can see that the thrust magnitude is 0.6 as we have seen previously. The non-linear increase and subsequent decrease of velocity indicates that a drag coefficient is in effect: this must be airdrag, since the other coefficients have been set to 0.

In the period of 30seconds to 35seconds, the player is walking forward on a normal floor surface.

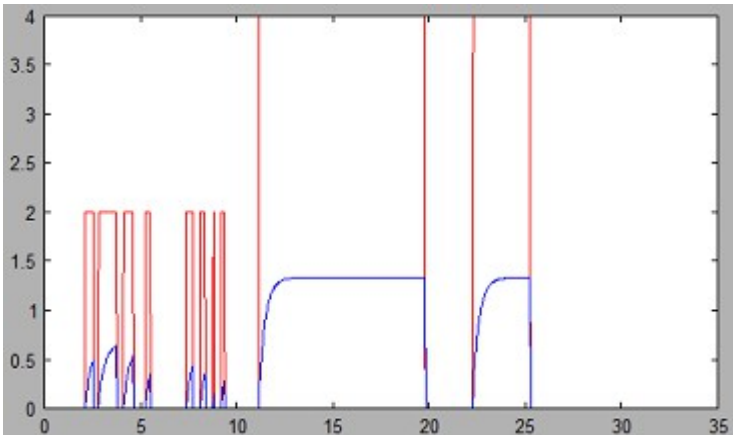
We can see that the thrust magnitude is 2.0 as we have seen previously. The very steep and linear increase in velocity is indicative of no drag coefficient being in effect. At approximately 33seconds when the player stops walking forward and the thrust magnitude drops to 0, we can see a corresponding leveling-off of velocity; also indicating no drag is in effect.

At 35seconds, the velocity begins to ramp back down as the player has returned to the non-floor surface; further exemplifying that the airdrag must be back in effect.

Therefore, this section should prove that **airdrag is the drag coefficient in play when the player is sliding on a non-floor surface.**

Staticdrag on normal surface

Next we will look at when the staticdrag coefficient comes into play. To do this, we will restore our other values to default and set the staticdrag value to "infinity" or otherwise some high value.



Drawing 5: Thrust/Velocity plot infinitestatic.m

staticdrag	999
surfdrag	3
airdrag	0.5

Table 6: walkplayer drag coefficients

Firstly we see that the initial 1/3rd of the graph have a peak thrust magnitude of 2 and the last two events have a thrust magnitude of 4. The actual in-game difference between these events is that the former thrusts were during walking and the last two were during running.

Now we will focus mainly on the longest apparent event, since all velocity responses to thrust impulses appear to have a similar result.

In the period of 11seconds to 13seconds, we see that the player velocity ramps up in response to the thrust in a similar fashion as we originally saw with player on normal surface. This implies that the "infinite" staticdrag coefficient is perhaps in no effect at this stage.

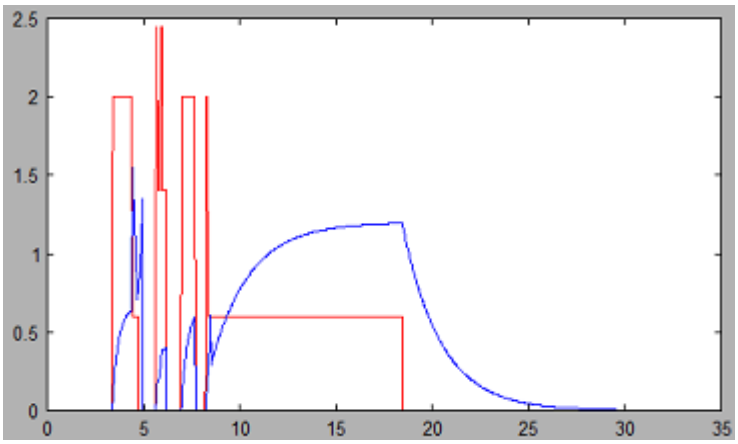
This is a little surprising since formal definitions of "static VS kinetic friction" seem to specify that "static" friction is in effect when an object is at rest, and effectively is the "budging" resistance that needs to be overcome before the object will move.

In contrast, at 20seconds when the player stops moving and the thrust drops to 0, the velocity also drops off to 0 virtually instantaneously. This implies that the "infinite" staticdrag coefficient is in effect at this point.

With some speculation, it should seem as though **surfdrag is in effect when the player is being actively accelerated by a force and that staticdrag is in effect when the player has no accelerating forces being applied**. It should also seem that staticdrag is independently used and not in any way used as an additive or scaling factor to the other drag coefficients.

Staticdrag on non-floor surface

Now that we have established a relationship between surfdrag and staticdrag on normal floor surfaces, we should check whether there is any relationship between airdrag and staticdrag on non-floor surfaces. For this test, we will continue with the same coefficients as the previous test, with staticdrag being effectively "infinite."



Drawing 6: Thrust/Velocity plot infinitestatic_nonfloor.m

staticdrag	999
surfdrag	3
airdrag	0.5

Table 7: walkplayer drag coefficients

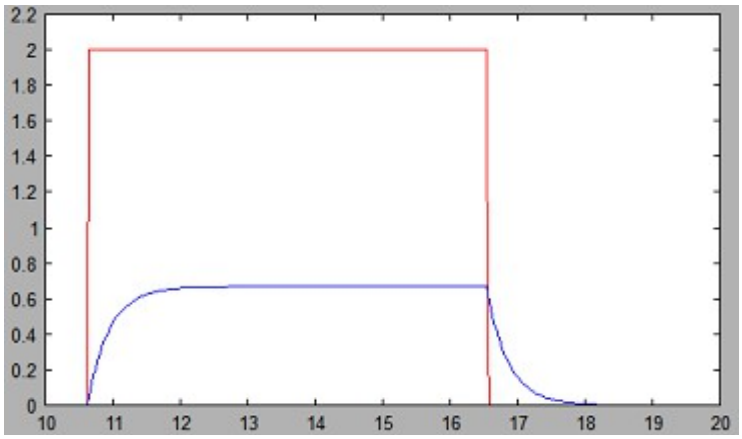
Within the first 8seconds or so, a test involving jumping was attempted, but the resulting data is convoluted with other player movements and so we will disregard this section.

In the period of 10seconds to 30seconds, we can see a rise and fall in velocity in response to thrust in a similar fashion as an earlier test of walking on non-floor surface, where we found that airdrag was the coefficient in use.

Note that unlike the surfdrag vs "infinite" staticdrag test, we are not seeing an immediate drop-off of the velocity when the thrust is removed. **This suggests that staticdrag has no effect when on non-floor surfaces, and by extension, as an object is moving through free space.**

No staticdrag

Revisiting our very first test, a walk on normal surface was repeated but with staticdrag removed.



Drawing 7: Thrust/Velocity plot nostaticdrag_walk_normalsurface.m

staticdrag	0
surfdrag	3
airdrag	0.5

Table 8: walkplayer drag coefficients

Of particular note is at roughly 17seconds whereby the velocity continues to approach 0 at a very slow rate. This is in contrast to the original sample (which included staticdrag) where a large portion of the velocity ramp down seemed to be clipped and set virtually instantaneously to 0.

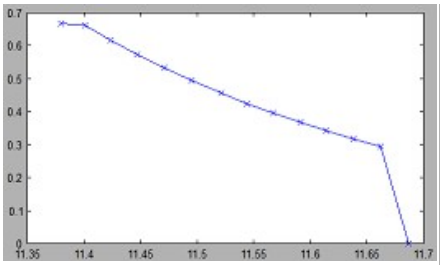
Further, this seems to emulate the effect of the Icy / VeryIcy surface flags. It is therefore assumed that **the Icy / VeryIcy surface flags may cut staticdrag down by some amount**, or perhaps in the case of VeryIcy, explicitly use a staticdrag of 0. This is simply an observation and the exact behavior of these surface flags is not examined in this particular stage of research.

Static Drag

The question remains whether staticdrag acts as a continuous resistance like airdrag and surfdrag, or if it serves as a threshold point to set velocity to 0. This is a particularly interesting question as the default value of 0.3 has a fairly abrupt effect even though it is a lower value than, say, surfdrag of 3 or airdrag of 0.5.

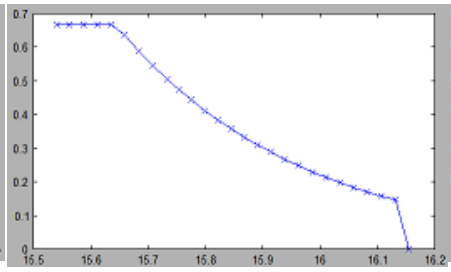
Therefore, three tests are performed with somewhat arbitrarily chosen staticdrag values of half, normal, and double. The thrust data is removed and we focus only on the velocity rampdown stage of the player stopping movement.

staticdrag	0.15
surfdrag	3
airdrag	0.5



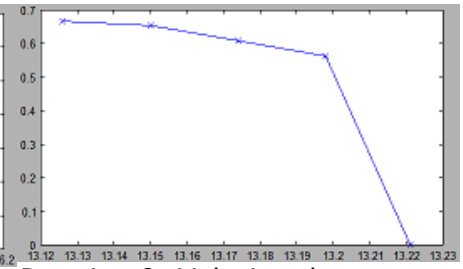
Drawing 9: Velocity plot halfstaticdrag.m

staticdrag	0.3
surfdrag	3
airdrag	0.5



Drawing 10: Velocity plot normalstaticdrag.m

staticdrag	0.6
surfdrag	3
airdrag	0.5



Drawing 8: Velocity plot doublestaticdrag.m

These results are fairly telling. Note that the actual datapoints are marked with an X in this series; this means that the **drop to velocity 0 is actually instant**, even though the line connecting the datapoints suggests a linear decrease.

Also, it is crucial to notice that the point at which **the drop to 0 velocity occurs is when the velocity has reached the staticdrag value**. From these results it should then appear as though staticdrag acts as a threshold value.

Dynamic Drag

Now that we have determined that staticdrag acts as a thresholding value and that airdrag and surfdrag act as a continuous resistance, it is important to discover how these dynamic drag values behave.

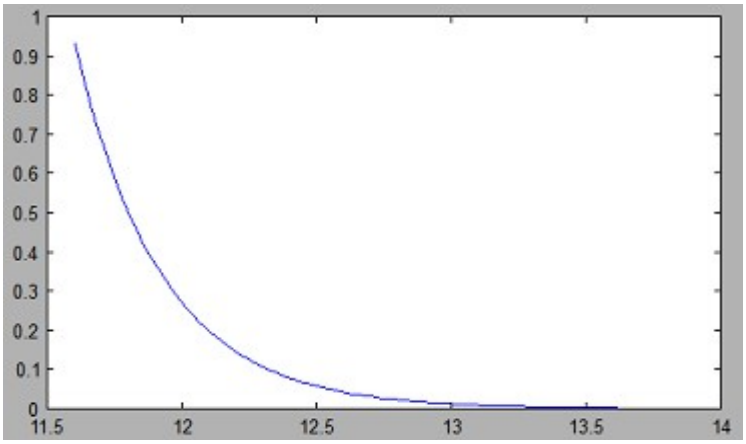
It is initially assumed that airdrag and surfdrag are applied in the same way since they have quite a large difference in value (walkplayer surfdrag is 6x airdrag) and a layman's observation in-game seems to correlate with a difference of this scale.

To perform the following tests, it will be useful to control the environment a bit more by ensuring we are starting our measurements with a fixed velocity value. Therefore, the "velocityresearch.cog" script within the Velocity Hallway will assign a velocity value directly on the player upon entering the sector.

In addition to starting with a fixed velocity, we will set the staticdrag to 0 to allow the player to come to rest purely from dynamic friction. The amount of time for this to occur will be compared across multiple tests with different values.

Raw data

Player velocity 1 on normal surface with default surfdrag of 3.



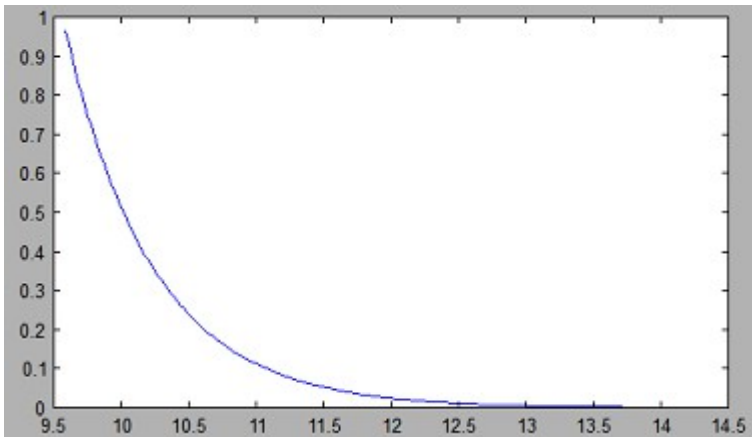
Drawing 11: Velocity plot surfdrag_normal.m

staticdrag	0
surfdrag	3
airdrag	0.5

Table 9: walkplayer drag coefficients

Subtracting the start/finish timestamps shows that it took roughly 2.193 seconds to go from a velocity of 1 to 0.

Player velocity 1 on normal surface with half default surfdrag of 1.5.



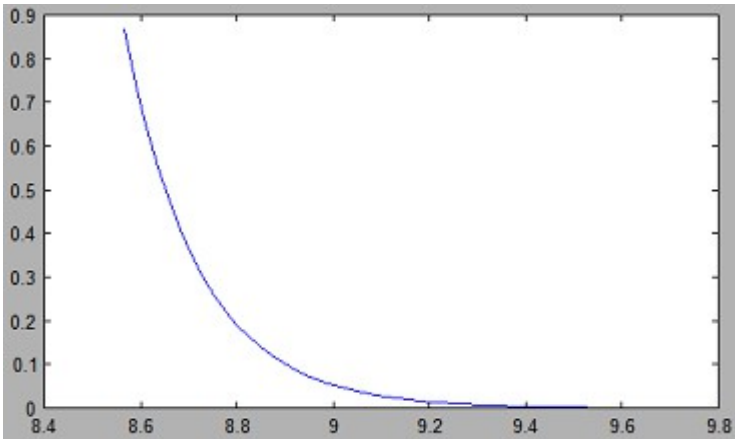
Drawing 12: Velocity plot surfdrag_half.m

staticdrag	0
surfdrag	1.5
airdrag	0.5

Table 10: walkplayer drag coefficients

Subtracting the start/finish timestamps shows that it took roughly 4.488 seconds to go from a velocity of 1 to 0.

Player velocity 1 on normal surface with double default surfdrag of 6.



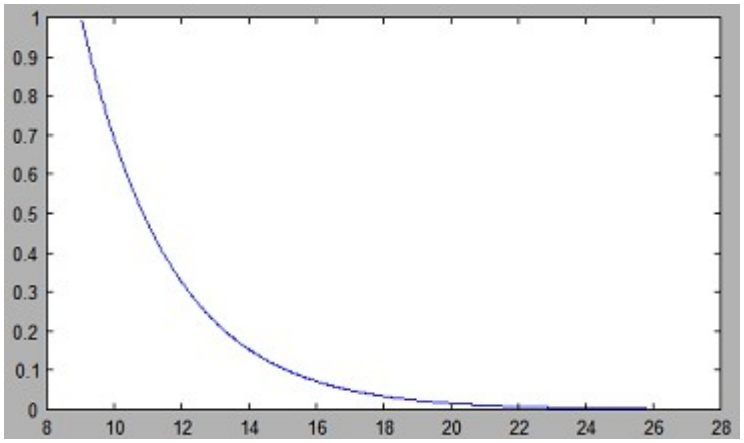
Drawing 13: Velocity plot surfdrag_double.m

staticdrag	0
surfdrag	6
airdrag	0.5

Table 11: walkplayer drag coefficients

Subtracting the start/finish timestamps shows that it took roughly 1.042 seconds to go from a velocity of 1 to 0.

Player velocity 1 on normal surface with 1/8th default surfdrag of 0.375.



Drawing 14: Velocity plot surfdrag_eighth.m

staticdrag	0
surfdrag	0.375
airdrag	0.5

Table 12: walkplayer drag coefficients

Subtracting the start/finish timestamps shows that it took roughly 18.309 seconds to go from a velocity of 1 to 0.

Let's bring our four test results into a table for comparison.

Inverse multiplier %	surfdrag	Time to reach 0 vel	Time % of normal
800.00%	0.38	18.31	836.00%
200.00%	1.5	4.49	205.00%
100.00%	3	2.19	100.00%
50.00%	6	1.04	47.50%

Particularly we are comparing the Inverse multiplier percentage against the Time percentage.

Considering each step above 100% of normal results in an increasingly higher and higher amount of time, and a 50% results in a lower than 50% time, it appears as though we have a non-linear response. If the result was linear then our Inverse multiplier and Time percentages would be matching.

Therefore, it should be safe to claim that **dynamic drag is applied as a function of the current velocity** rather than a static value.

Summary

Analyzing one step of the raw data of the normal surfdrag instance, we find these values:

	Velocity	Timestamp
v0	0.869554	11.624001
v1	0.814772	11.645

dv	dt	da	**da/v0**	da/v1
-0.0548	0.021	-2.60952381	**-3.001**	-3.203

Table 13: Velocity reduction as a function of time

In bold is the nominal calculation which appears to represent the walkplayer's surfdrag coefficient of 3.

It is presumed that v1 represents the newly-calculated velocity for the current frame, and v0 represents the existing velocity from the previous frame.

When the current velocity is subtracted from previous velocity (dv) and derived to find the acceleration, we find a value of -2.6. When this is divided by the previous velocity then we find a value of 3 which matches our surfdrag coefficient.

Since our answer was found using the calculated acceleration (da), this proves that dynamic drag is applied as a force (not directly affecting velocity, necessarily).

Reversing the math, we find the following:

v0	surfdrag	-Acceleration	dt	dv	v0	v1
0.869554 x	3 =	-2.608662 x	0.021 =	-0.054781902 +	0.869554 =	0.814772

Table 14: Proof of velocity update as a result from dynamic friction

The final formula is provided in the FINDINGS section of this document.

Mass

Gravity

There are actually at least two tutorials/articles on Massassi that state that mass affects how fast a thing falls due to gravity. This is in direct contradiction to classical mechanics.

A test can be set up and performed rather quickly where two things with wildly different masses can rest upon an impassible surface over a chamber. Activating a switch to change the surface to passable and detaching the things simultaneously should show whether they appear to fall at the same rate or not.

The test was conducted and found that a throwcrate with a mass of 0.1 and a throwcrate with a mass of 1000.0 fell at the same rate and landed at the same time.

The player was maneuvered to push against each crate and it was therefore confirmed that the low mass crate moved easily and the high mass crate moved with more difficulty. Although this result was expected, article(s) on Massassi have humorously indicated that (except for the “falling rate due to gravity”) the purpose of mass was unknown.

Orientation

Introduction

Orientation, typically represented in degrees as a pitch/yaw/roll (“PYR”) triplet, is used to tilt and rotate things within the game world.

There are a variety of ways to manipulate the orientation of a thing, or even specific meshes within a thing's 3d model. For example: a rotational velocity makes powerups spin, the player's aim adds pitch to the entire upper half of the model and complicated “troop transport” pathways can be defined.

For the most part, a thing's position seems to be finalized to a single XYZ vector. Any additional physics or offsetting is applied to the existing value in order to calculate the new one. It should be noted as well that the order of operation doesn't matter when adding vectors, eg. Position + A + B == Position + B + A.

It doesn't appear to be this straight forward with orientation. For example, a spinning fan which has a rotational velocity seems to correctly spin regardless if the fan is rotated to be horizontal or vertical. This implies that, in this example, the rotational velocity updates another internal orientation value which is different from the thing's overall orientation; thereby allowing the base transformation to effectively change the coordinate space of the next.

In further comparison to the simplicity of adding vectors, the XYZ components also can be added in any order, eg. {P.x + A.x + B.x} {B.y + P.y + A.y} {A.z + B.z + P.z} produces the correct output of P + A + B.

However with orientation, the order is extremely important as each component effectively transforms the entire forward/up/right coordinate space. Applying pitch then yaw produces a completely different result than applying yaw then pitch.

Internal Representation

It should be stated that it is not known for certain how JK stores orientation internally.

It could store multiple PYR triplets which are concatenated dynamically.

Or, like a position vector, it could store a single set of forward/up/right vectors and incrementally update them as PYR changes are applied.

Finally, it's possible that quaternions are used either as a storage medium or during the concatenation process.

Of particular curiosity is the fact that there is no “GetThingRot” in COG, yet there do exist “GetThingLVec”, “GetThingUVec”, “GetThingRVec”. This is a significant indicator that JK may not retain PYR values internally. Further, it would be more efficient to store finalized look/up/right vectors and use them directly rather than dynamically concatenate multiple PYR triplets.

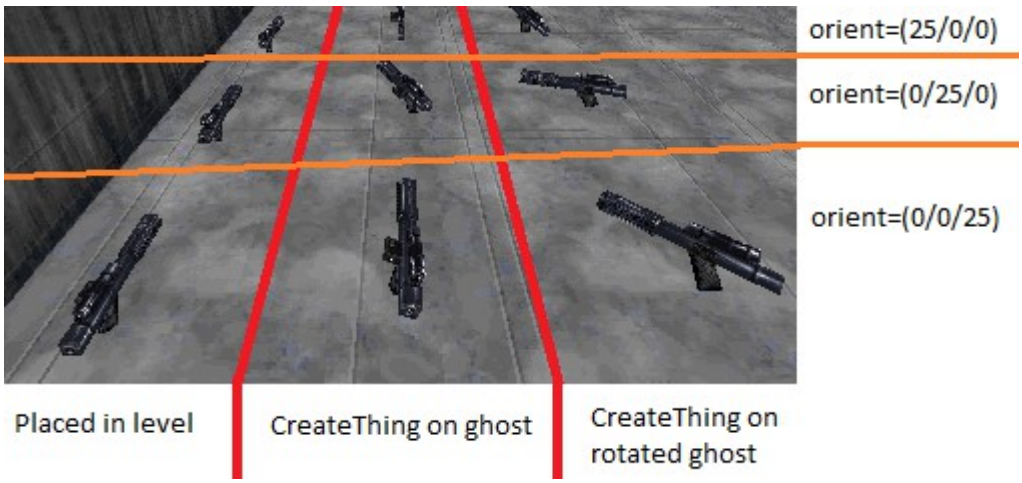
It should be said, of course, there does exist a “GetThingRotVel” in COG, so it can be assumed that this value is indeed stored; although it is not necessarily indicative of the overall thing's orientation storage medium.

Order of Operations

When a thing template has an “angvel” property defined (and/or presumably from SetThingRotVel) then the axis of rotation seems consistent with its orientation as defined in the JKL. In other words, if a powerup (spinning along yaw) is placed at some awkward angle in the level then the rotation of the powerup still appears to rotate around the rendered model's up vector.

“orient” template property

Let's start by creating multiple thing templates with varying “orient” properties, and create the thing in a variety of ways.



These results are immediately revealing.

First of all, all three strifles in the left column, which were placed into the level in JED, are unaffected by the “orient” property in the template. **It would seem that the PYR in the JKL's thing placement overrides the “orient” property in the template.**

Secondly, all three strifles in the middle column, which were created via COG “CreateThing”, all have an orientation corresponding to the thing's template. **Things created dynamically use the “orient” property in the template.**

Finally, all three strifles in the right column, which were created via COG “CreateThing” on ghosts which have a 45 degree yaw rotation, have a final orientation of the ghost orientation plus the “orient” property. **Things created dynamically on an already-oriented thing inherit both orientations.**

CreateThing vs CreateThingNR

It has been postulated that “NR” stands for “No Rotate”, so it seems appropriate to test the effect when discussing orientation. Here we have repeated the scenario of dynamically creating things with both an “orient” property defined in the thing template, and a 45 yaw rotation defined on the ghost.

On the left is “CreateThing” and on the right is “CreateThingNR”. As we see, there is no difference.

Attachment

Thing attachment to other things and level surfaces is an integral part of the physics system. It is used to switch between air to surface drag coefficients, facilitate stable movement of things on elevators and determine if a character is able to jump; among many other purposes.

Flags

There are a number of flags that determine or otherwise affect attachment behavior.

Surface Flags

Of particular note is the Floor surface flag (0x01).

Physics Flags

Of particular note is the AttachToFloor (0x40), AttachToWall (0x80) and AlignOrientationToSurface (0x10) thing physics flags.

The Attached (0x100) flag is involved. DataMaster describes this as being briefly cleared when attaching to a surface. Testing confirms this. It was speculated that perhaps this flag is cleared when a thing is being transitioned to its insert offset position but the flag was not manipulated when cycling between crouching/standing, when insert offset transitioning is occurring. Of further note is the 0x100 flag remains set regardless if attached to a surface or not (eg. jumping up does not clear it; only when landing is it briefly cleared).

DataMaster indicates that 0x800 flag is used to apply gravity while attached to a surface. It is reported that clearing this flag, for example, can allow a player to stand still while walking on walls.

DataMaster indicates that 0x4000 is used to determine if a thing can still be affected by blast forces when attached to a surface.

DataMaster indicates that 0x200000 is cleared when attached to a surface.

There may be additional physics flags which are related to attachment in some way; perhaps some of the otherwise currently “unknown” flags or ones involving water surfaces.

Attach Flags

There is an enormous difference in the definitions of attachment flags between DataMaster and JKSpecs; so much though that it almost seems like they are defining a different set of flags.

Hex	Dec	Used with	Purpose
0x1	1		Thing is attached to a surface.
0x2	2	0x4	Thing will be given the lookvector of the object it's attached to.
0x4	4		Thing will be given the same movement given to the object it's attached to.
0x8	8	0x4	Thing will be attached to a base object by its relative position.

Table 15: DataMaster Attach Flags Specification

0x0	Not attached
0x1	Attached to a world surface
0x2	Attached to the face of a thing
0x4	Attached to a thing, but not a certain face of that thing
0x8	No Move relative to attach thing

Table 16: JKSpecs Attach Flags Specification

It has been noted in other circumstances that DataMaster tends to be a more accurate resource. Therefore, subsequent testing will be focused mainly on confirming the DataMaster definitions.

Weapon Flags

Finally, there are weapon flags which result in attachment such as in the case of sticky raildets or sequencer charges.

The 0x80 flag specifies that the weapon thing will attach to level surfaces.

The 0x800 flag specifies that the weapon thing will attach to things.

Insert Offset

It is understood that the INSERT OFFSET, defined within a 3do header, is used to effectively place a character above the ground. This is done so that the character's overall collision radius can be reduced.

Considering the fact that a character on a non-floor surface appears to sink in, presumably to the collision radius size, indicates that there must be a direct relationship between insert offset and attachment.

What is curious is that when a player is standing on the ground, his actual collision sphere is situated above the ground. In other words, the collision sphere is not touching and yet the character no longer falls due to gravity.

An initial question to be answered is whether insert offset is used as a separate measure to determine a floor surface under the character, or if the character's actual collision sphere must intersect with the floor surface before detecting the collision and offsetting the character up. This can be easily checked by setting the insert offset Z component to a large value.

```
INSERT OFFSET  -0.000050   0.000000   0.3
```

The results of this test are revealing in several ways.

First of all, it definitely appears that the player's collision sphere is used to detect initial floor collision. As soon as the player lands on the floor, he is quite quickly (but not instantly) shifted upwards to the target insert offset point. This took a number of frames; perhaps a quarter of a second. This transition felt like a linear speed rather than any dynamic acceleration.

Secondly, it was noticed that crouching seemed to circumvent the effect of the modified insert offset entirely. The crouch height appeared exactly as it normally would. It is suspected that this Z offset is simply 0. Also noted is going into and out of crouching had the same effect as when initially landing: a fairly quick (but not instant) transition between a presumed 0 Z insert offset and the one defined in the 3do.

Next we will explore how the other components of the insert offset vector are used. An arbitrary -0.1 was added to the X component:

```
INSERT OFFSET  -0.100050   0.000000   0.3
```

This appeared to have absolutely no effect. The test was run again, adding a fairly sizable 0.3 to Y component; also with no effect. It is suspected perhaps the full INSERT OFFSET is used in other circumstances (such as when inserting a thing into the level editor) and **only its Z component is used, as a bit of a hack, to handle the character height** VS collision sphere size.

It was also noticed that the increased height allowed the player to seamlessly walk onto other surfaces that normally would have required jumping, and this resulted in the same sort of quick (but not instant) transition to restore the nominal height. This helps to explain some of the 'magic' behind JK's physics look&feel.

Another interesting test is to set the Z component of insert offset to 0.

```
INSERT OFFSET  0.000000   0.000000    0.000000
```

This behaved exactly as you would imagine, I suppose. The player was sunken into the floor slightly, presumably in the same fashion as standing on a non-floor surface (although this was not checked). Interestingly, however, going into crouch actually raised the player/camera view ever so slightly. Note that this *did* appear to be an instantaneous switch, unlike the linear interpolation described earlier. It is suspected that the node positions in the crouch animations are the reason behind this, but this also has not been checked at this time.

Detecting Detachment

It is well understood that the COG command DetachThing can be used to arbitrarily cause the player to detach from a floor surface. Other triggers, such as jumping, are also known to result in a detachment. For example, if the player enters a sector with an upward thrust then the player is not accelerated upwards until he jumps. Conversely, if the thrust sector's floor is not flagged as floor then the upward acceleration begins immediately.

What needs further clarification is how exactly JK determines that a thing is no longer attached to a surface such as when the player walks off a platform. Conversely, how JK handles the seamless transition of attachment when walking along a pathway made of multiple individual surfaces (eg. the long spiral walkway along the perimeter of Canyon Oasis).

Surface Flags

A test area was created with a button that toggles the floor flag on/off of the surface under the player.

When the player is completely stationary and the floor flag is cleared then nothing initially happens. As soon as the player attempts to move the lack of floor flag is detected and he sinks in. Likewise, when sunken in and completely stationary, and the button is pressed to set the floor flag, nothing happens until the player attempts to move. At this time the “landing” is detected and the floor attachment occurs.

The test was repeated, but this time the “passable” adjoin flag and the “impassable” surface flags are also toggled.

This, too, resulted in nothing happening until the player attempted to move.

An implementation that should result in the same behavior may be that, while a thing is attached, to perform a raycast from the thing position down to his feet. **If an intersection point is found then set the target position to that point plus the insert offset height.**

~~If an intersection was not found (or is beyond the insert offset height) then clear the attachment.~~

Further, to match JK's behavior, this test should only be performed if the thing is moving.

Descending From Height

When walking down a slope, stairs, or even falling off a slight height, it appears as though the player is never actually detached. The falling animation does not play, no sound is made, etc. This has not been confirmed via tests but for now is assumed to be the case that an actual surface attachment does not occur. Rather, a seamless reattachment to the new surface is performed.

The previously suggested idea of performing a raycast to the player's feet in order to detect detachment is not good- it means that every small incline will cause the player to continuously detach. The assumption, then, is that JK extends the range of possible seamless reattachment further than just the player's feet. The extent of this can be checked quite easily by making several pits of varying depth and observing what happens when the player gingerly moves over the side of them.

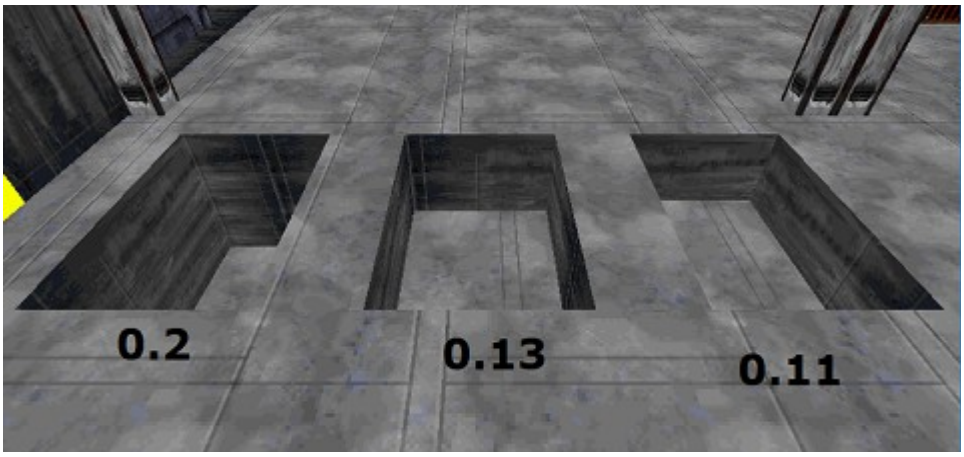


Illustration 5: Various pit depths

The right two pits were chosen to have a depth each of slightly less and slightly more than the player's insert offset of ~0.12.

It should be noted, curiously, that detachment did NOT occur in either of the two right pits; even when running over them. The left 0.2 depth pit did result in detachment.

The other curiosity observed is when the player transitions to attachment to the bottom of the pit, it is not a completely smooth transition. Instead there seems to be a bit of a "bump" point that is transitioned to first, then a following transition to the final point. This was observed even in the 0.11 depth pit.

The tests were continued, bringing the pit depths closer and closer to the point where falling VS seamless reattachment (“transitioning”) occurred.

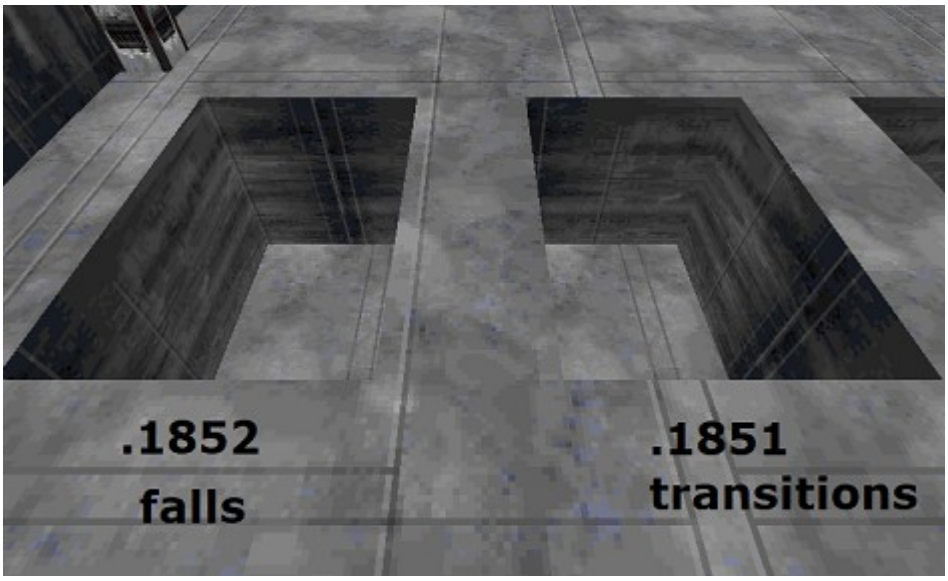


Illustration 6: Detachment vs Transition heights

This is pretty interesting because of the number of decimal places. Once a grid snap of 0.0001 was reached, the tests were stopped; so the real changeover point likely has even more precision.

This implies that the changeover point is likely calculated rather than hardcoded. Stemming from the fact that the player's insert offset of 0.120114 has so many decimal digits, it seems plausible that some standard scale factor against this value could result in a changeover point with many digits. It should seem that the scale factor would be 1.5, however this produces 0.180171 which is close but we have found the changeover point to be a little higher than that.

It seems possible, then, that 1.5x might be correct, but with an extra measure of fudge factor. For example insert offset Z times 1.5 (scale factor) plus 0.005 (fudge factor) would result in 0.185171 which lays between our changeover point as experimentally measured.

A way to confirm the relationship with insert offset and confirm a static fudge factor value is simply to change the player's insert offset and repeat the measurements.

The player's insert offset height was changed to 0.06 and, optimistically, the pit depths were adjusted to values calculated by our proposed formula.

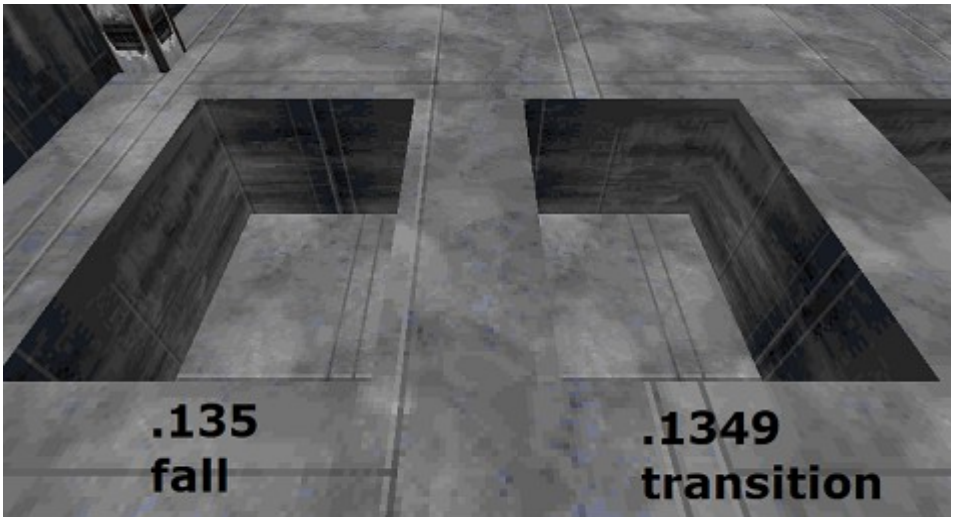
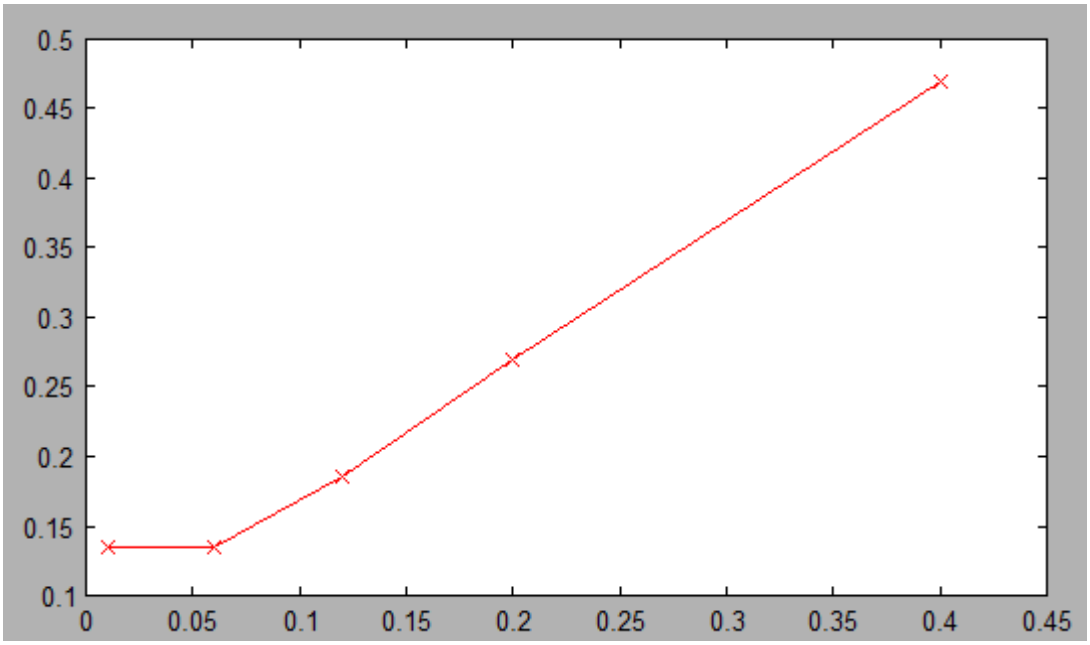


Illustration 7: Detachment vs Transition heights; insert offset 0.06

Well, our calculated pit depths of around 0.095 were incorrect. However, we were right that changing the insert offset changes the changeover point.

If we blindly divide this pit depth and our insert offset, 0.135 / 0.06 = 2.25 scale factor, which we previously thought might just be 1.5. Since we have confirmed the changeover point IS calculated, at least in part, by the insert offset then we either have an additional value being factored in prior to scale, or it is simply not a linear relationship.

Although rather tedious, it may prove revealing to repeat the test with additional data points and plot the curve we get to show the nature of the relationship between insert offset and the resulting detachment changeover point.



Drawing 15: Insert offset vs detachment changeover

A couple of interesting observations can be made. Firstly, at 0.06 and 0.01 insert offset, the changeover point behaved identically (as noted by the horizontal line at the left). Secondly, the initially apparent non-linearity occurs between 0.06 and the standard ~0.12, yet when increasing up to 0.2 and 0.4 the response appears perfectly linear.

These two observations indicate that the player's size or movesize must be in effect since it is defined as 0.065. This would explain both phenomenon in the left portion of the plot.

Taking a look at the numbers toward the right hand side of the plot, we find that an insert offset of 0.2 has roughly a 0.27 changeover point. Further, an insert offset of 0.4 has roughly a 0.47 changeover point. It is then probable that the real changeover point is 0.265 and 0.465, respectively. **This would imply that the raycast distance is simply equal to the greater of either insert offset or size/movesize, plus size/movesize.**

```
RaycastDist = max(Radius, InsertOffsetZ) + Radius;
```

This formula would explain the minimum ~0.13 we see at the low end, and the additive relationship at the upper end.

It should be noted that we have been dealing with pit depth; in other words from one floor attachment level to another. Therefore our raycast distance is actually from the player's feet. **If the origin of the raycast is at the player's position then the insert offset should be added again.** This time, probably without the max(Radius, InsertOffsetZ) logic; if we really want to emulate as if the raycast begun at the player's real feet position.

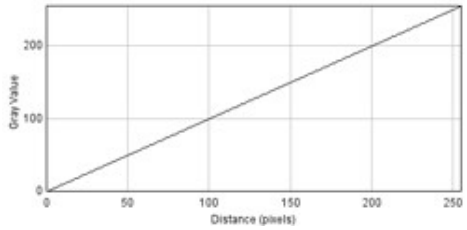
Lighting and Color

Color effects are used to tint the overall screen to achieve various effects. For example, underwater tints the screen a sort of bluish or aquamarine. Picking up powerups causes the screen to flash with a bit of color. Force blinding causes an overbrightening effect, making it very difficult to see.

Color Correction

To test whether JK has gamma correction we will generate a grayscale test gradient:

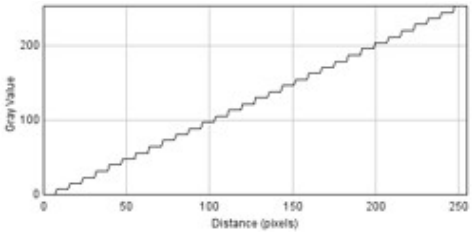
Illustration 8: Original grayscale gradient



Drawing 16: Profile plot of original grayscale gradient

Using Mat16, this is converted to a 16-bit MAT file. The result was converted back to a 24-bit RGB bitmap for comparison purposes.

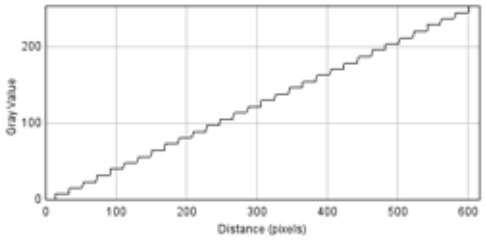
Illustration 9: Grayscale gradient after conversion



Drawing 17: Profile plot of quantized grayscale gradient

The in-game result:

Illustration 10: Grayscale gradient in-game



Drawing 18: Profile plot of rendered grayscale gradient

It does not appear as though a non-linear color transformation is being applied, by default.

HOWEVER: It should be noted that there is a "Gamma" hotkey.

Color Effects

An initial attempt at guessing the algorithm for color effects has proven unfruitful. A more scientific approach is necessary. For this, we will sample the final rendered colors of a known color with various effects applied.

Due to incompatibilities on modern PC hardware, the following tests may be done using the software renderer rather than hardware acceleration. This could result in some deviation of how the color effects are applied. It also means that 16-bit MAT textures are not supported. Therefore, an 8-bit paletted MAT texture was generated by the following image using the colormap *09fuel.cmp* as was originally selected for the Smith Dev research level.

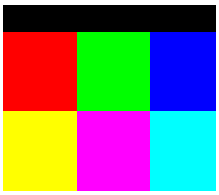


Illustration 11: RGB test texture

Since the *09fuel.cmp* palette does not necessarily include the exact color equivalents as desired, the magenta and cyan regions will be ignored. A baseline in-game rendering of the RGB test texture is as follows:

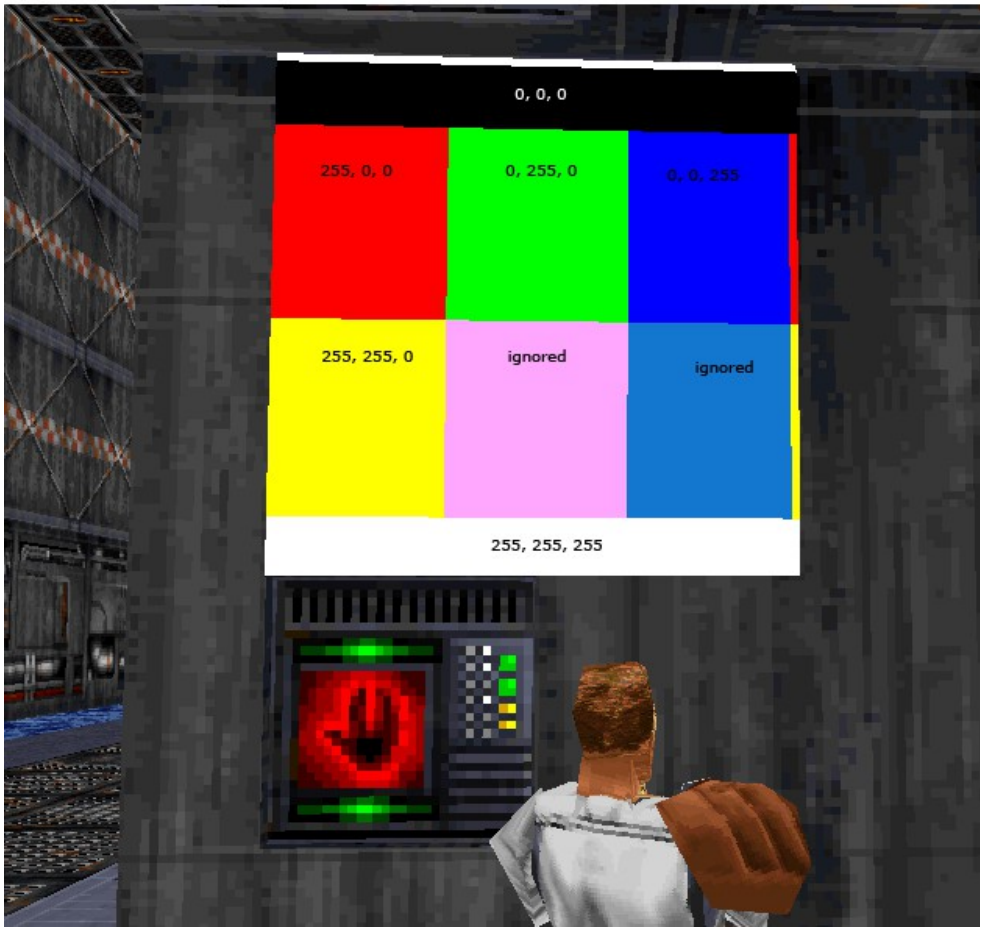


Illustration 12: Baseline rendering of RGB test texture

Again, it should be noted that the magneta and cyan regions, which do not have an exact match within the choice of colormap, are replaced with an approximation and should therefore be ignored.

All remaining regions are rendered at the exact pixel values as specified in the original texture. This means that measuring the pixel values of these regions when color effects are active in future research should assist in determining the exact mechanisms of which the effects are applied.

COG ColorEffect

The COG commands NewColorEffect/ModifyColorEffect are quite powerful in that they allow specification of RGB values for three different types of effects, plus a fourth control value. It is suspected that sector tint, "dynamic tint" and "dynamic add" are likely implemented in a similar fashion as [some] of these effect modes. A useful feature for reverse engineering using this ColorEffect is that the commands last indefinitely until explicitly canceled; in contrast to AddDynamicTint which dissipates rapidly once initiated.

The first challenge in deciphering the COG ColorEffect is in its conflicting specification from JKSpecs and Datamaster.

ColorEffect according to JKSpecs

Of initial note is the fact that JKSpecs only defines ModifyColorEffect and not NewColorEffect.

Here is the definition in its entirety:

```
Does some cool stuff with colors. Mixing, tinting, etc...

Use: ModifyColorEffect(int _filterR, int _filterG, int _filterB, flex
tintR, flex tintG, flex tintB, int _addR, int _addG, int _addB, flex
fadeintensity);
```

ColorEffect according to DataMaster

Contrarily to JKSpecs, DataMaster specifies that ModifyColorEffect is not used by LEC and instead behaves identically to FreeColorEffect. However, its NewColorEffect definition does describe the same parameters but in a very different way:

```
Creates a color_effect that will be displayed until FreeColorEffect()
removes it. Syntax:

effect_handle = NewColorEffect(R1_int, G1_int, B1_int, R2_flex,
G2_flex, B2_flex, R3_int, G3_int, B3_int, flex_black);

Group 1: The first RGB group is a 0 or 1 integer that tints the level.

Group 2: The second RGB group is a 0-2 flex that adds a deep tint to
the level.

Group 3: The third group is a 0-290 integer that adds a color to the
screen.

flex_black: This is a 0 to 1 flex that adds black to the screen. A
value of 0 would be fully black.
```

Initial discussion

Firstly, the discrepencies between JKSpecs and DataMaster should be determined.

Indeed, the few LEC COGs that utilitize NewColorEffect do appear to different units for different modes.

kyle.cog:

```
blindingEffectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 200 + 8 * rank, 210 + 10 * rank, 200 + 8 * rank, 1.0);
```

force_seeing.cog:

```
effectHandle = newColorEffect(0, 1, 1, 0, 0, 0, 0, 0, 0, 1.0);
```

This would imply that the first set of RGB triplets has a range of 0..1 whereas the last set of RGB triplets has a range of 0..290 (in the case of *210 + 10 * rank*, considering items.dat defines jedi_rank to have a range of 0..8).

For the third set of triplets it would initially seem that a range of 0..255 is more appropriate. It is possible that the COG is designed such that the effect winds up saturating at a jedi rank lower than maximum, if the values are clamped to a maximum of 255. A sensible initial test would be to identify the real ranges that these parameters have an effect.

An initial test using 0's for all triplets results in no observable or measurable effect on color rendering.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 0, 0, 0, 1.0);
```

The next tests will experiment with various values for the “add” effect.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, X, X, X, 1.0);
```

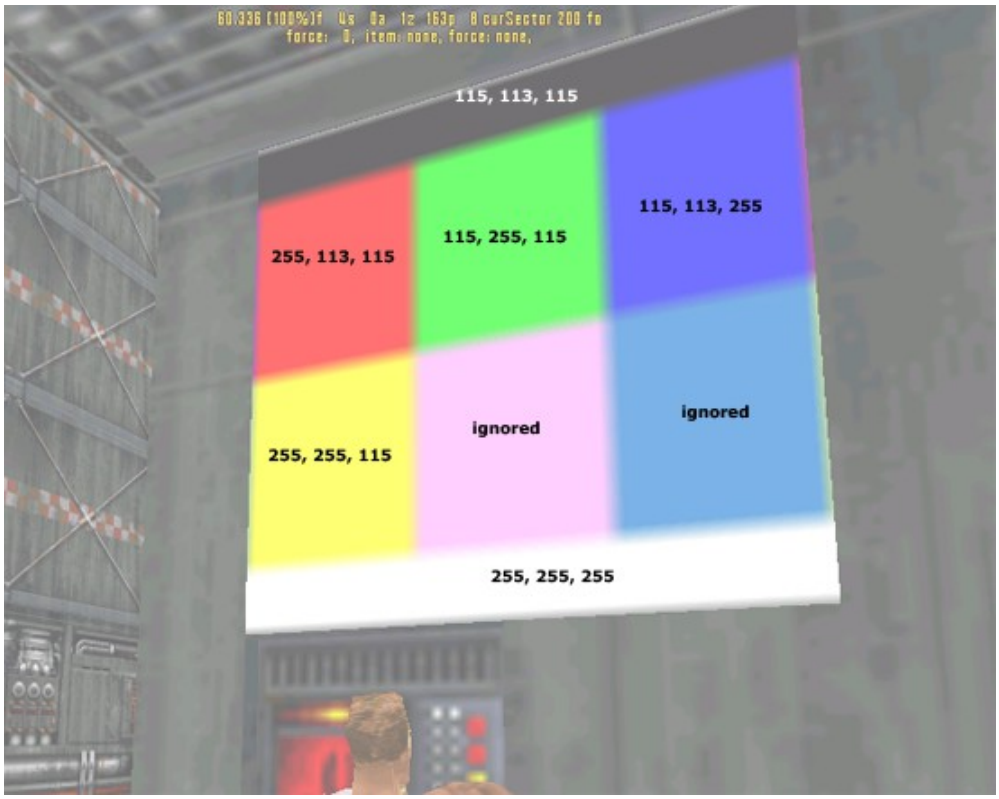


Illustration 13: ColorEffect Add 128,128,128

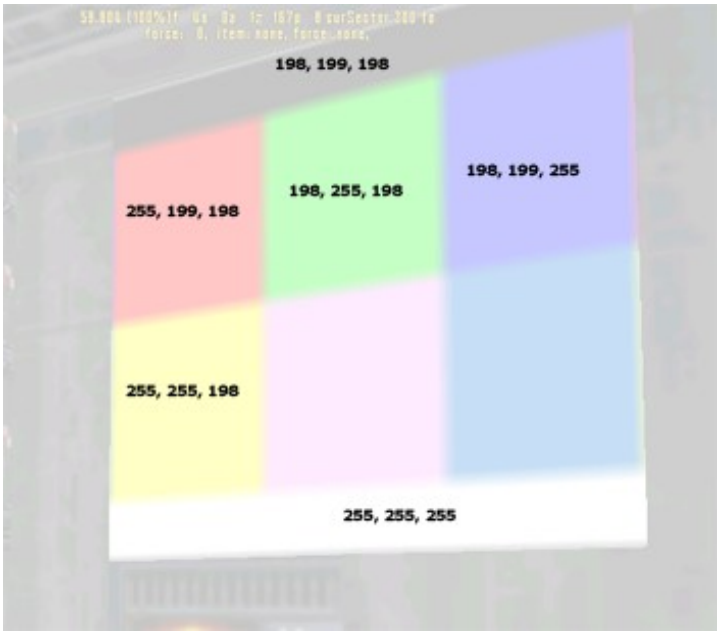


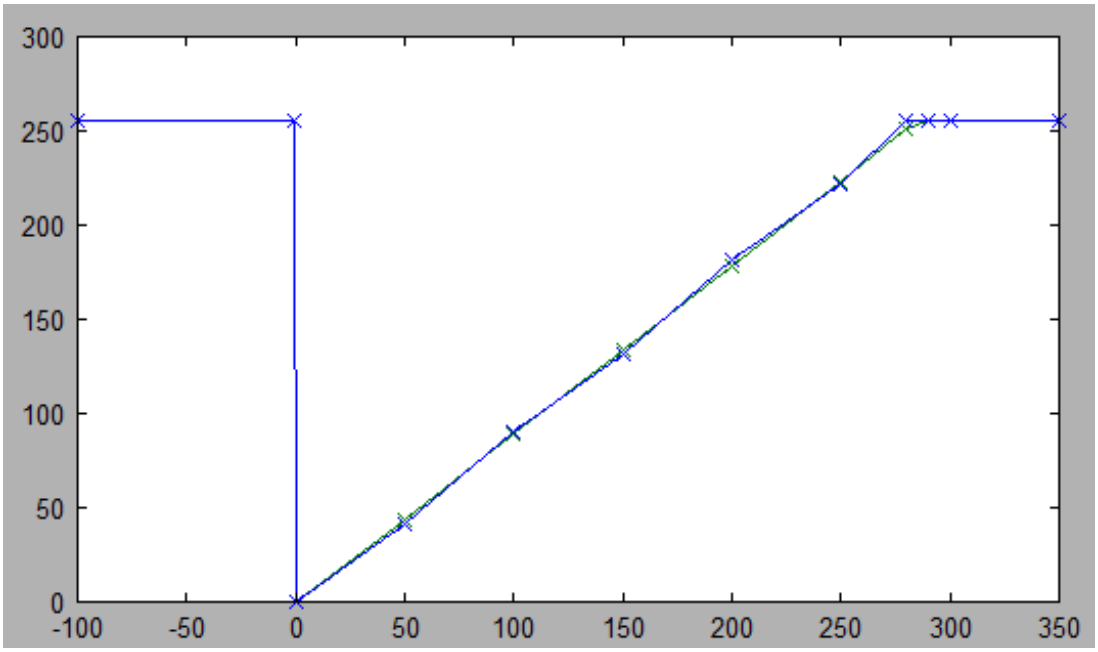
Illustration 14: ColorEffect Add 220, 220, 220

From these initial runs, it is a little curious that the components tend to have a slight deviation.

Add

Perhaps the most useful approach will be to try multiple runs at consistent intervals (using the same value for red, green blue) and plot the rendered RGB data points measured from the black region of the color test texture as a graph to identify the valid range and linearity.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, X, X, X, 1.0);
```



Drawing 19: ColorEffect Add response

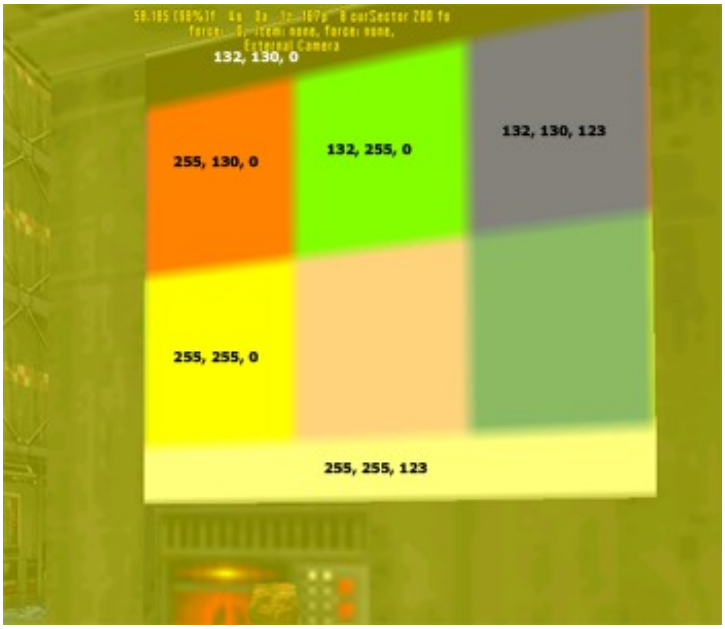
Several observations can be made.

Firstly, any negative number results in the screen becoming fully white. This also occurs with values past the maximum range. DataMaster's purported 290 maximum looks to be accurate.

The response curve appears essentially linear, however some slight deviation across color channels can be seen which is somewhat perplexing. One possible explanation may be that the effect is added in an alternate colorspace, eg. HSV (Hue/Saturation/Luminance), and that approximations or floating point error when converting between colorspaces results in the slight variance. On the other hand, if a 16-bit color mode is in use, these could be precision errors within a 565RGB color sceme: this seems like a likely scenario considering the R / B channels showed a value of 198 while the G channel showed a value of 199 in the ColorEffect Add 220 test.

An interesting effect of Add can actually result in a reduction of other color channels.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 145, 145, 0, 1.0);
```

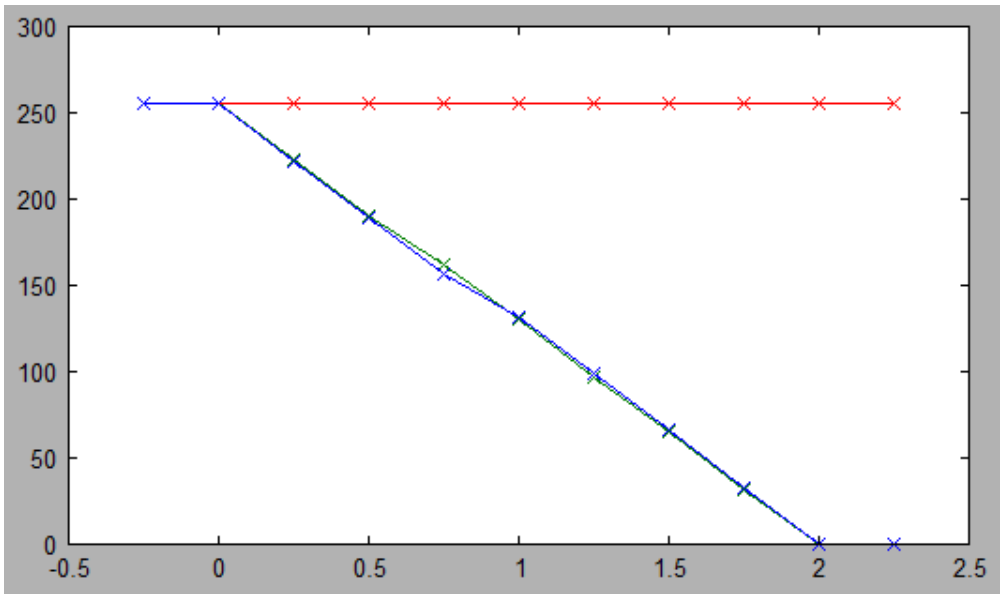


Notice how the white region is 255, 255, 123. Perhaps a good next test will be on multiple different shades of gray.

Tint

Next, we shall continue on to the "tint" component of ColorEffect (second set of RGB triplets). A similar test will be done using increments both outside and within the presumed range of 0..2. Since tint is expected to work more as a scaling rather than additive value, the rendered RGB measurements will be taken from the white region of the color test texture. Finally, since tinting of any grayscale value appears to have no effect, we will focus on the Red channel only.

```
effectHandle = newColorEffect(0, 0, 0, X, 0, 0, 0, 0, 0, 1.0);
```



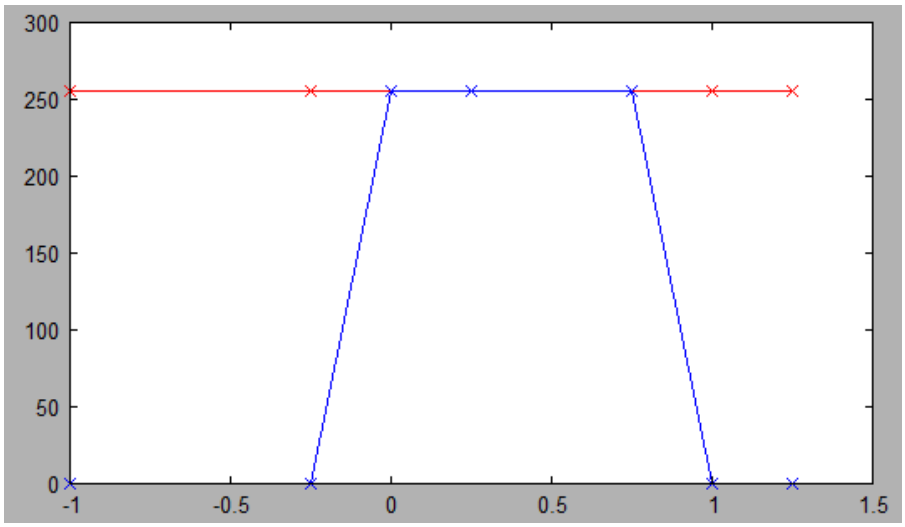
Drawing 20: ColorEffect Tint response (red only)

We can see confirmation of DataMaster's indicated valid range of 0..2.

Filter

Finally, we will get an initial impression of the "filter" component of ColorEffect (first set of RGB triplets). A similar test will be done using increments both outside and within the presumed range of 0..1. Since filter is expected to work more as a scaling rather than additive value, the rendered RGB measurements will be taken from the white region of the color test texture. Finally, since filter of any grayscale value appears to have no effect, we will focus on the Red channel only.

```
effectHandle = newColorEffect(X, 0, 0, 0, 0, 0, 0, 0, 0, 1.0);
```



Drawing 21: ColorEffect Filter response (red only)

This appears to be treated as a boolean value. Any value less than 0 and any value equal or greater than 1 turns on the effect.

The following tests use a Filter mask of 1 for all three channels except for one.

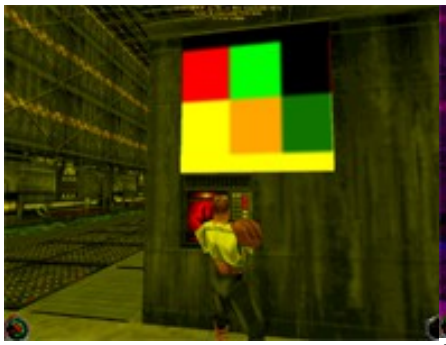


Illustration 15: ColorEffect Filter 1/1/0

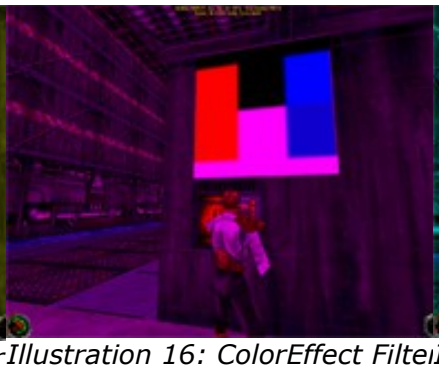


Illustration 16: ColorEffect Filter 1/0/1

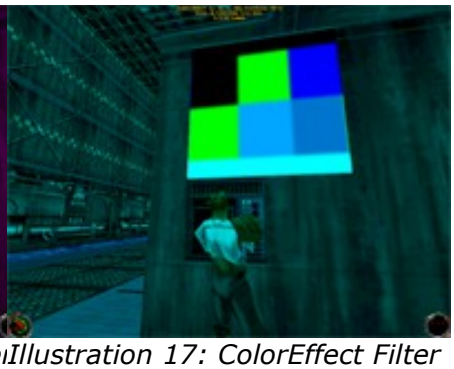


Illustration 17: ColorEffect Filter 0/1/1

In all examples, it can be noted that the channel which is turned off appears completely black in the corresponding color test texture. For example, in the middle example where the green channel is 0, the green square on the texture is black.

This strongly suggests that the Filter works very simply by performing a bitwise mask of 0x00 or 0xFF depending on the boolean value of the Filter channel (or, in terms of floating point, multiplying the component by 0.0 or 1.0).

Brightness

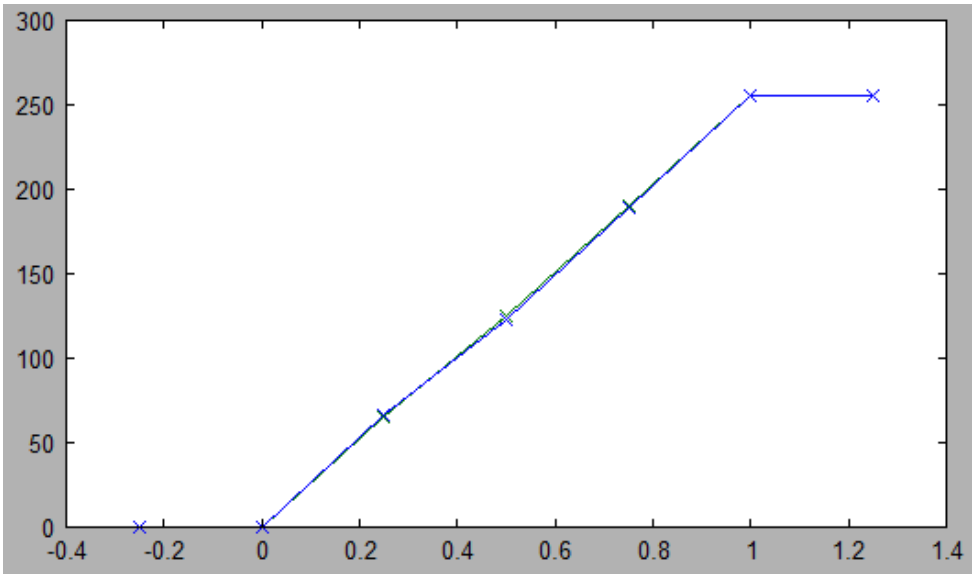
The descriptions of the very last value of ColorEffect have wildly differing descriptions. So far, DataMaster has been more detailed and more accurate, so it is initially suspected that this value is an overall screen brightness rather than a delay value.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

Indeed, this results in a completely black screen.

Let's try a few values and plot the RGB channels as with previous tests.

```
effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 0, 0, 0, X);
```



We find that the effect is linear and is clamped between 0..1.

It should also be noted that an over 1.0 value did not boost the brightness of dark regions. In other words, it is not possible to add light to the scene; only blackness as indicated by DataMaster.

Filter vs Tint

A visual inspection of the difference between Filter and Tint (at full strength of 2.0) is presented here.

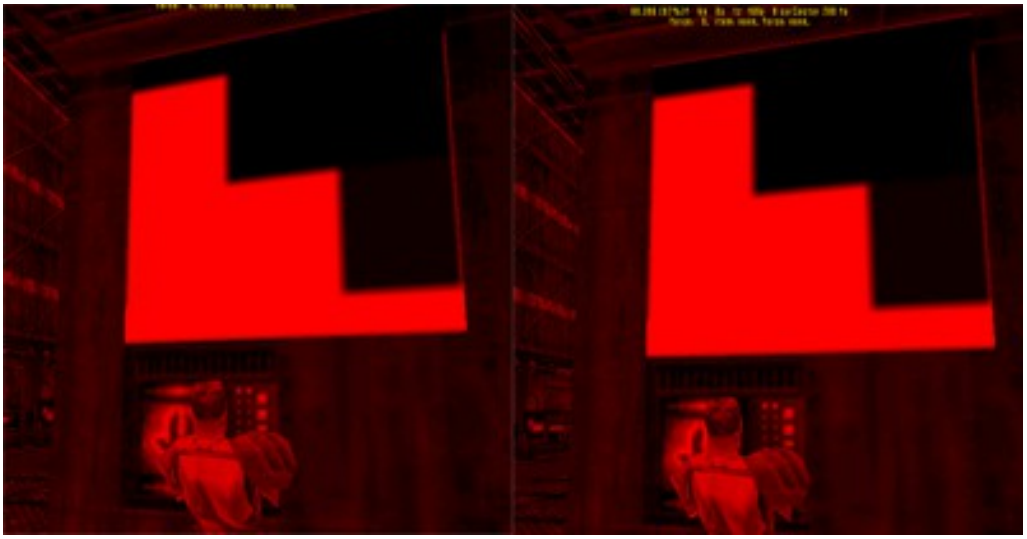


Illustration 18: ColorEffect Filter vs Tint (full strength)

It is quite plain to see that these are identical renderings.

This is still not enough to confirm their behaviors since the interactions with other factors (such as other ColorEffect or IR mode) are not yet tested at this stage.

Filter and Tint combinations

A test was conducted with a red color filter, with a full green/blue tint.

```
effectHandle = newColorEffect(1, 0, 0, 0, 2, 2, 0, 0, 0, 1.0);
```

As suspected, the final rendering was a completely black screen.

The same test was repeated but with only a 50% tint strength.

```
effectHandle = newColorEffect(1, 0, 0, 0, 1, 1, 0, 0, 0, 1.0);
```

This also resulted in a completely black screen.

Another test was done with a 50% red/green tint.

```
effectHandle = newColorEffect(1, 0, 0, 1, 1, 0, 0, 0, 0, 1.0);
```

This resulted in an identical rendering as with just a red filter.

It is apparent that Filter and Tint can interact with each other in some way.

Filter/Tint and Add combinations

Next, a test with red color filter and full green add.

```
effectHandle = newColorEffect(1, 0, 0, 0, 0, 0, 0, 290, 0, 1.0);
```

This resulted in a completely green screen, which indicates that Add is unaffected by Filter.

Next, a test with full red tint and full green add.

```
effectHandle = newColorEffect(0, 0, 0, 2, 0, 0, 0, 290, 0, 1.0);
```

This resulted in a completely green screen, which indicates that Add is unaffected by Tint.

It is apparent that Add is simply added as a final step and is not modified by any preceding operation.

Add and Brightness

~~We will test with full green Add and zero brightness.~~

~~effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 0, 290, 0, 0);~~

~~This resulted in a fully green screen.~~

~~Now we will test with a half yellow Add and quarter brightness, and measure the pixel value in the white region.~~

~~effectHandle = newColorEffect(0, 0, 0, 0, 0, 0, 145, 145, 0, 0.25);~~

Illustration 19: ColorEffect half yellow Add with quarter brightness

~~The white portion of the test texture measures as 165, 162, 33.~~

~~Applying the math manually: 255,255,255 * 0.25 + (145,145,0 * 255 / 290) = 191,191,63.~~

~~This result is perplexing. The measured value overall is lower than our calculation would indicate it should be.~~

Tint and Brightness

We will test with full red Tint and half brightness.

```
effectHandle = newColorEffect(0, 0, 0, 2, 0, 0, 0, 0, 0, 0.5);
```

Illustration 20: ColorEffect Red tint with half brightness

Interestingly, the half brightness value seems to have no effect here. The bright red areas are fully saturated at 255, 0, 0.

An initial guess may seem to be that Tint is applied after Brightness, and that the algorithm that Tint uses is able to restretch the rendered colors to full range.

Filter and Brightness

We will test with red Filter and half brightness.

```
effectHandle = newColorEffect(1, 0, 0, 0, 0, 0, 0, 0, 0.5);
```

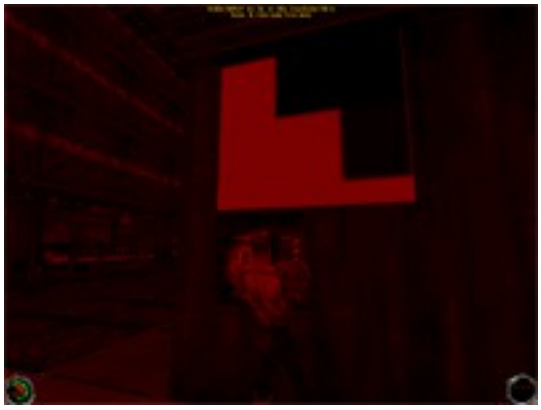


Illustration 21: ColorEffect red Filter with half brightness

Now we are starting to identify some different behaviors. The brightest red areas are only saturated with 123, 0, 0.

This definitely indicates that Filter is applied prior to, and is subsequently affected by, Brightness.

Filter and Brightness and Tint

The guess that Tint may be applied after Brightness and causes the dynamic range to be stretched back out to full range will now be tested.

```
effectHandle = newColorEffect(1, 0, 0, 2, 0, 0, 0, 0, 0, 0.5);
```

Illustration 22: ColorEffect red Filter, half Brightness, full red Tint

The brightest red regions are saturated at a value of 255, 0, 0.

This strongly suggests that the guess of Tint causing an expansion of dynamic range may be correct. It appears to have nullified the effects of Filter/Brightness.

IR Mode

Initially it would appear as though IR mode effectively makes the level render as fully-lit. However, the method which this mode is activated has s suggestive parameter:

item_irgoggles.cog:

```
EnableIRMode(0.3, 1);
```

force_seeing.cog:

```
EnableIRMode(0.3 + 0.05 * rank, 1);
```

Considering jedi_rank ranges from 0..8, it can be seen that under standard LEC COGs the first parameter ranges from 0.3 to 0.7.

Dynamic Light Attenuation

Perhaps a good way to determine JK's attenuation values for dynamic lights is to set up a test bed with the following method:

- Create a totally dark room with a dynamic light in it of a certain light value
- Use a grid texture or some way to map out distance from the light
- Take a screenshot at each of these predefined distances and measure a final rendered pixel value
- Record the distance → light level values into a plot
- Repeat this process with a dynamic light of different light value

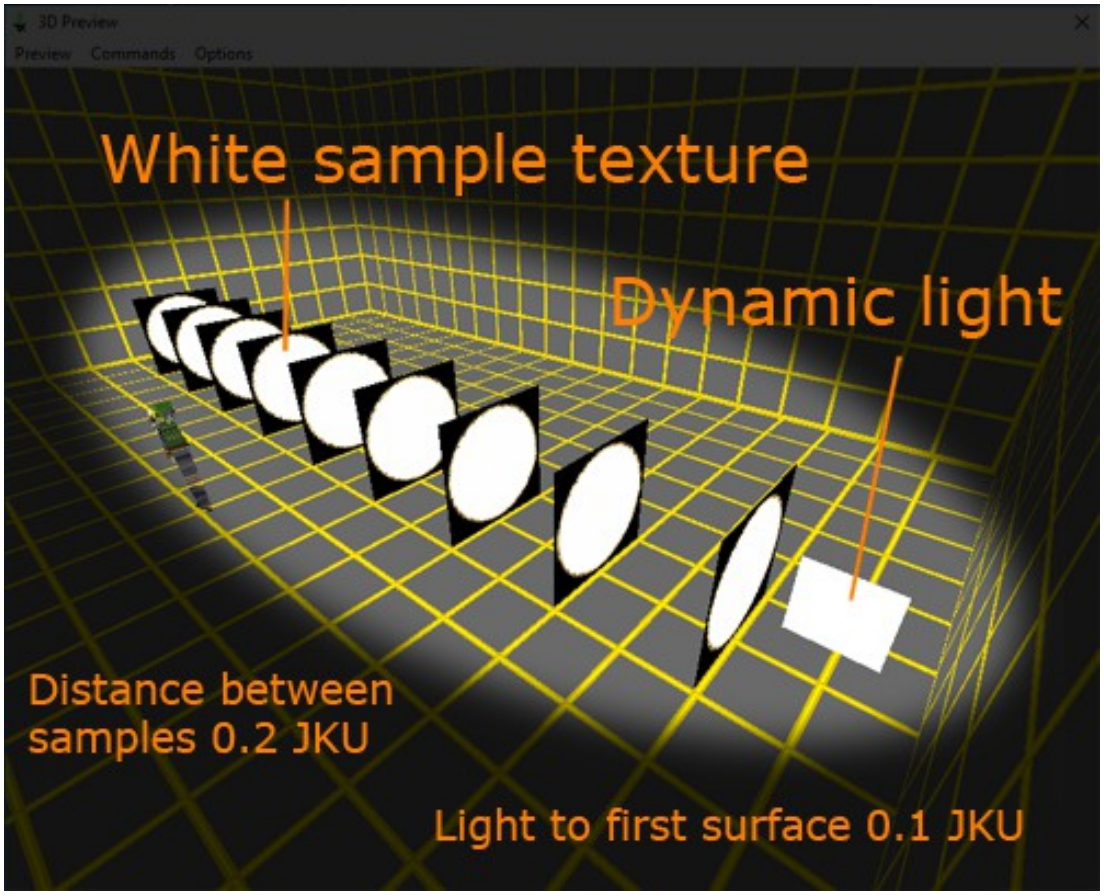


Illustration 23: Dynamic light attenuation test bed

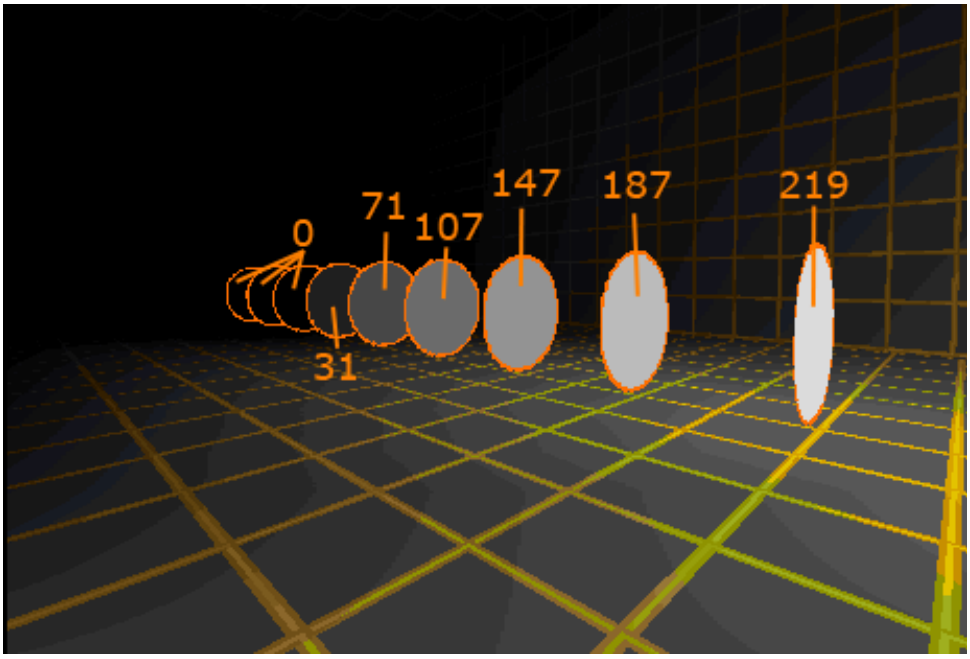
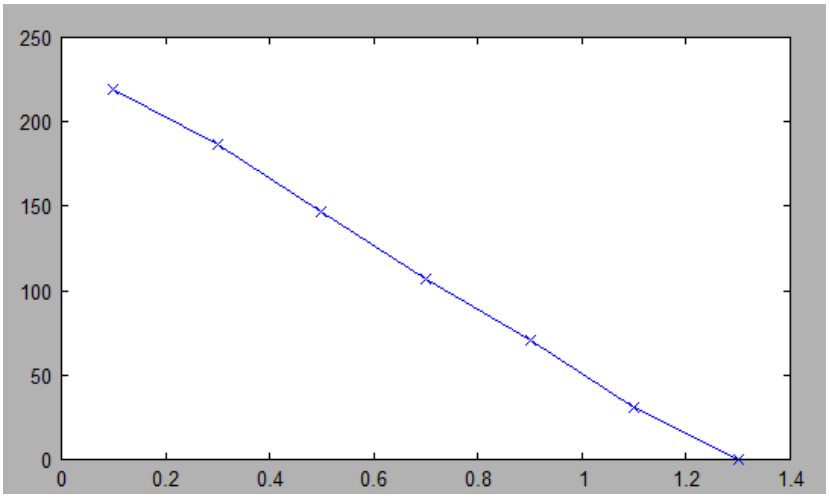


Illustration 24: Pixel values for dynamic light 1.0



Drawing 22: Pixel values over distance plot for light 1.0

Well; I'm a bit surprised. That looks strikingly linear.

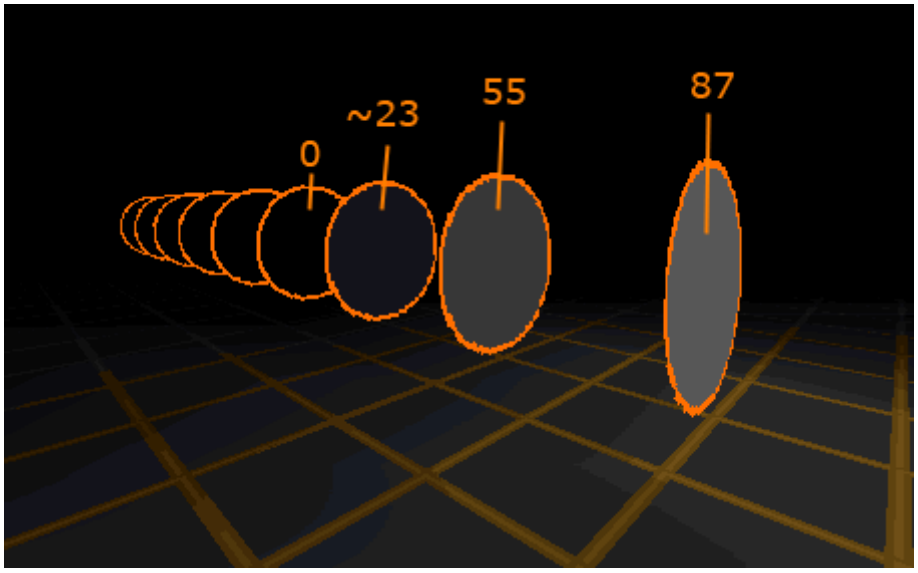
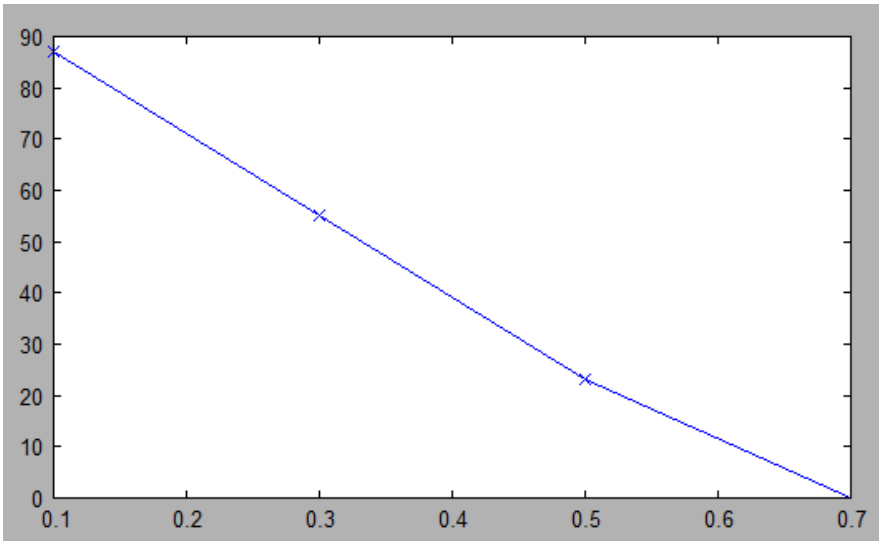


Illustration 25: Pixel values for dynamic light 0.5



Drawing 23: Pixel values over distance plot for light 0.5

There aren't many data points here, but we'll still include these results in our final analysis.

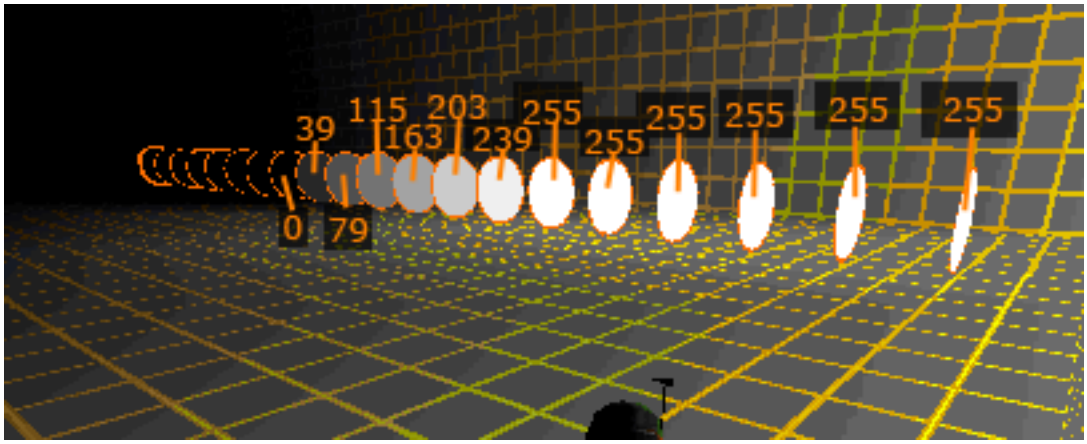
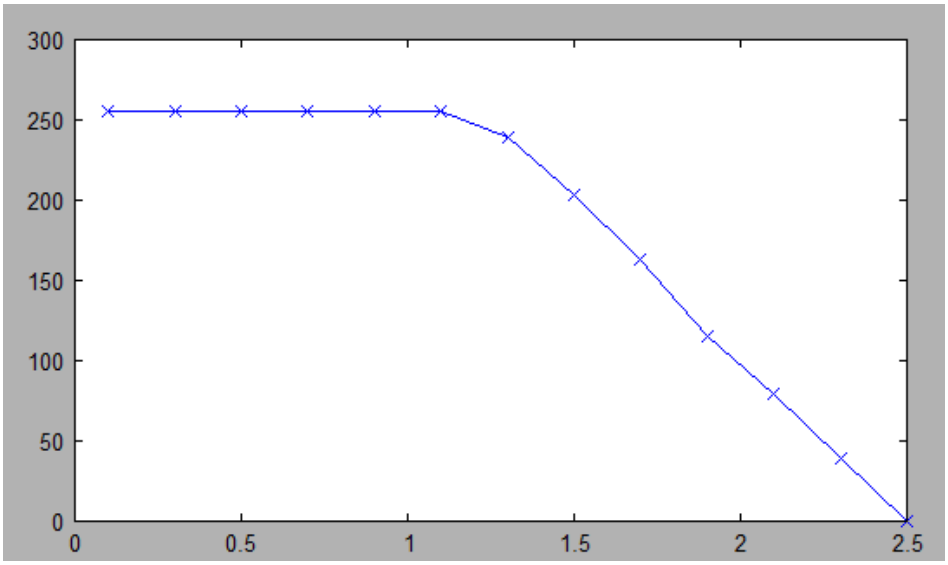


Illustration 26: Pixel values for dynamic light 2.0



Drawing 24: Pixel values over distance plot for light 2.0

As we can see, the first handful of targets are essentially overbrightened and clamped to maximum brightness.

Once the attenuation begins, it still appears to be linear.

Revisiting the data from the first example, with light level 1.0, we can notice something peculiar:

distance	0.1	0.3	0.5	0.7	0.8	1.1	1.3
light	219	187	147	107	71	31	0

Table 17: Pixel values over distance for dynamic light 1.0

If we make some assumptions then a light level 1.0 with linear attenuation might result in a radius 1.0 illumination area. This means we would expect to see a light level of 0 at the 1.0 distance mark; but we don't. In fact a light level of 31 is reasonably bright or at least clearly not black. It is close, however.

If we walk up the template tree for “light1.0” we don't find that any radius or movesize is defined anywhere; so there doesn't appear to be a source for, say, a minimum light radius. Perhaps there still is a hardcoded default in the engine. This would have the effect of essentially not beginning the attenuation until some minimum distance from the light source has been reached. If present, this would explain the discrepancy between our prediction and the measured result.

Now this gets really interesting. The light source has been moved VERY close to the first target surface. The X coordinate of the surface is 0.8 and the X coordinate of the light source is 0.8001.

With a dynamic light level of 1.0, the surface is still only lit to 219 which is what we originally got. We know its possible to get 255 because we got a whole slew of them with dynamic light 2.0. So even though the attenuation seems to be linear, there still seems to be something fishy going on.

Now we will generate a table of light levels to their actual rendered values, using this extremely close light source:

light level	0.1	0.3	0.5	0.7	1	1.3
pixel value	0	47	99	147	219	255

Table 18: Maximum illumination ability of dynamic light values (to surface center)

Before continuing, the light source was moved to still be 0.001 from the target surface, but aligned on Y and Z to match one of the surface vertices.

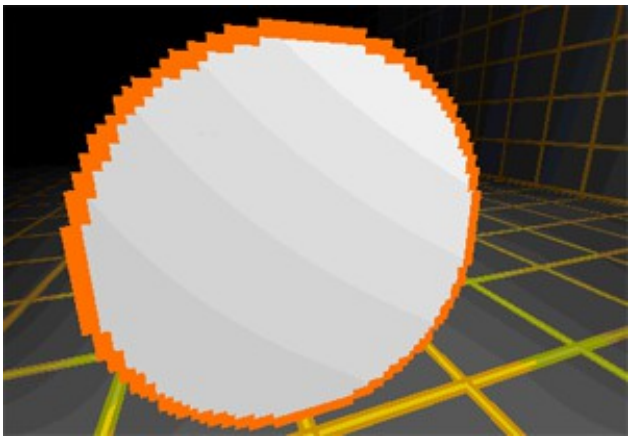


Illustration 27: Light 1.0 moved to vertex

The brightest area at upper-right has pixel value of 239. Presumably if this texture was white instead of transparent at the upper-right corner then it would render pixel value 255.

Since our light source in all previous tests has been aligned to the surface center, each vertex appears to have been given effectively the same distance weight and so the surfaces are rendered very evenly but at a slightly artificially lower level due to the slightly longer distance from vertices to surface center. This may invalidate some of our data collected thus far.

The surface texture has been changed to include a small white square in the top right so we may continue our tests and hopefully minimize this distance error.

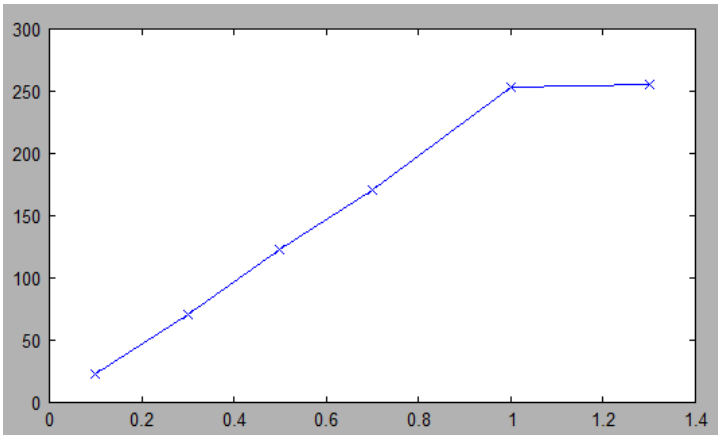
Now the light 1.0 produces a pixel value of 253. It's possible that the 0.001 distance might be causing this to be lower than 255; if that's the case then we could potentially rule out any possible "minimum radius" as speculated earlier.

Let's regenerate our maximum illumination table using this vertex and texture adjustment.

light level	0.1	0.3	0.5	0.7	1	1.3
pixel value	23	71	123	171	253	255

Table 19: Maximum illumination ability of dynamic light values (to surface corner)

Plotted out, that looks like:



Drawing 25: Light level to maximum illumination ability

So thankfully, the active region still appears linear. At this point we're still not sure if there is an inherent "illumination ability" limit based on light level, or if the maximum brightness we're able to achieve out of a light is still solely due to its attenuation strength over distance.

In other words, is a light's actual maximum intensity scaled by the light level, in which case full brightness is *literally* impossible? In this case we might find that the attenuation factor is the same for all lights.

Or is the light value basically always 1.0, with the light level only affecting the strength of attenuation such that very low light levels have such strong attenuation that full brightness is just *practically* impossible?

We actually may have all the data we need already in order to identify this.

Looking back at the table of data we just generated, notice how light level 1.0 results in 253; just a couple down from 1.0 * 255 = 255. Now look at light level 0.1 results in 23; just a couple down 0.1 * 255 = 25.5.

Firstly, this suggests that there is a direct, linear relationship between light level and maximum brightness.

Let's make a new table that shows the difference between theoretical maximum brightness and what we measured.

light level	0.1	0.3	0.5	0.7	1
predicted	25.5	76.5	127.5	178.5	255
measured	23	71	123	171	253
difference	**2.5**	**5.5**	**4.5**	**7.5**	**2**

Table 20: Light level variances from predicted

This is a little tough to stomach because the difference values dance around quite a bit, but the important thing to note is that they are not *trending*. They are not steadily increasing nor decreasing with changing light levels. Therefore, although the error is a little unnerving and it might be nice to get a bunch more data points to confirm, it seems to be relatively safe to say that it is a constant difference regardless of light level.

Remember, we're assuming that the 0.001 distance and fact that the other surface vertices are not illuminated to the same level is what's dragging the value down slightly from the predicted maximum.

From this initial analysis, it seems that there is **a standard attenuation factor used for all dynamic lights and the light level only affects the starting value.**

PARTICLES

The particle effects are something that seems to have never really been documented well by the community.

NOTE: the behavior of upwards floating particles, falling spark particles, or rotating particle clouds seems to be a function of the physics flags / vel / angvel attributes of the particle thing itself rather than applying to the individual particles! This seems to be an important note to make during the introduction of the particle system research, even though this discovery had not been made yet during the following initial experiments. Put succinctly, the particles positions themselves are local to the coordinate space of the particle thing.

Initial Experiments

To begin, we take the first particle template defined by JK, +whitecloud, and clear the typeflags to hopefully get a good baseline.

```
+partest          none

orient=(0.000000/0.000000/0.000000) type=particle timer=0.200000
typeflags=0x00 material=00gsmoke.mat range=0.020000 rate=128.000000
maxthrust=30.000000 elementsize=0.007000 count=128
```

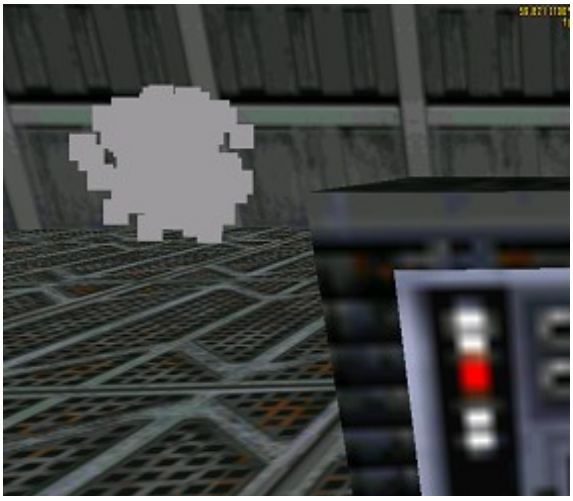


Illustration 28: Simple +partest particle template

It looks like there are probably 128 particles and they appear to be all created instantly within a small radius. They do not move nor fade in/out nor change color. After the 0.2 second delay, all particles immediately disappear. Each time +partest is created, the particle positions appear to be randomized within the creation space.

Let's try again but increase the range from 0.02 to 0.1

```
+partest          none

orient=(0.000000/0.000000/0.000000) type=particle timer=0.200000
typeflags=0x00 material=00gsmoke.mat range=0.10000 rate=128.000000
maxthrust=30.000000 elementsize=0.007000 count=128
```

Illustration 29: +partest with larger range

As expected, the creation radius increased but otherwise behaved identically to the first test.

Decay

The next test conducted involved increasing the timer to 1.0 and reducing the rate to 16. As expected, the particle field stayed in existence for longer, but interestingly changing the rate seemed to have no effect.

DataMaster indicates that the particle flag 0x08 should cause the rate value to determine how quickly to dissipate the particles after the timer expires rather than removing them all immediately.

After setting the 0x08 flag, we find that indeed after 1 second, the particles slowly start disappearing.

Also of possible interest is creating another +partest does not affect the previous one(s).

When the particle thing is destroyed 0.1 second after creation, the entire particle effect is removed instantly.

When the particle thing is destroyed 1.5 seconds after creation (which is after the timer expired but before the engine has cleaned up all particles), the entire particle effect is removed instantly.

This suggests that **the particle thing itself holds particles**, rather than the engine having a global particles list. Further, **the thing itself is not destroyed when the timer expires** but rather waits until all particles have expired (either instantly like in our original tests, or after they have decayed).

Note there seems to be **a lower limit to the rate parameter.** Setting it to 0 resulted in the same decay rate as 16. However setting it to 128 definitely resulted in a faster decay rate than 16. *As with other parameters, setting to 0 may be ignored, resulting in a default being used. This should be re-tested with a low value like 0.1, but this has not been done at this time.*

Creation

So far we've only seen effects where all particles are created instantly. In fact, cycling through 28 of the standard JK particle templates **does not appear that there even is a mechanism to have particles be created dynamically in any way other than instantly when the thing is created.** It seems the only impression of a growing effect is done by the original particles created in a small radius and moving outward.

It was noticed that *minsize* is often declared larger than *elementsize,* but smaller than (and similar in scale to) *range*.

Illustration 30: range=0.11 minsize=0.09

This result pretty plainly shows that ***minsize* and r*ange* define the minimum and maximum radii within with particles may spawn**; resulting in some pretty cool spherical effects.

Following up with the minsize/range experiment, yawrange and pitchrange were added into the mix.

Illustration 31: pitchrange=35 yawrange=35

Now this is cool. It's basically a slice of the exterior surface of a sphere; like a curved wall of particles. This really helps to reveal the logic behind particle placement.

It would seem that the **particle creation point involves randomly picking a value between the minimum and maximum radius and picking values +/- within the pitchrange and yawrange.**

To further isolate the pitchrange/yawrange behavior, the values were set to some extremes.

Illustration 32: Several activations with pitchrange=5 yawrange=170

The fact that the particles wrap in an *almost* full circle while leaving a gap indicates that the **yawrange/pitchrange values must be multiplied directly with a random value between -1 / +1**. So technically a pitchrange of 180 is actually a 360 degree range of coverage.

It should be noted that testing with pitchrange=0 worked the same as providing no pitchrange at all, or basically 180 (full 360 degree coverage). Setting pitchrange=0.1 did produce a thin band. It is assumed that yawrange also works in this way.

Coordinate space

An interesting test will be to see if a particle cloud follows a moving particle thing or if the particles seem to be created in global world space.

Illustration 33: Particle thing attached to moving thing

It absolutely appears as though all **particle positions are local to the particle thing's position**. This is evidenced by the fact that attaching the created particle thing to this moving bacta tank results in the particle cloud following the bacta tank.

However, a very peculiar effect was noticed with *some* particle templates where the effect cloud would continue going off in an original direction after the bacta tank had switched directions. The *+force_heal* effect shown in the screenshot exhibited this behavior.

Since this only seemed to occur with some effects, it seems likely that when the timer expires then the thing attachment is either broken or the particle thing otherwise ceases to process.

What's strange about this is the bacta tank is being moved via MoveToFrame and it's been thought that things attached to elevators, etc. have their positions directly updated rather than applying an acceleration/velocity to the thing. So perhaps either our integration paradigm of storing velocity differs from that of JK's, or perhaps the particle engine itself uses an alternative velocity calculation.

MORE RESEARCH IS NEEDED ON THIS TOPIC

Cel Animation

For this initial test we picked *00ramp1.mat*, a material used in sparks effects which are known to change color over time. This was then paired with the 0x04 flag which DataMaster reports to cause each particle to pick a random cel index in the material.

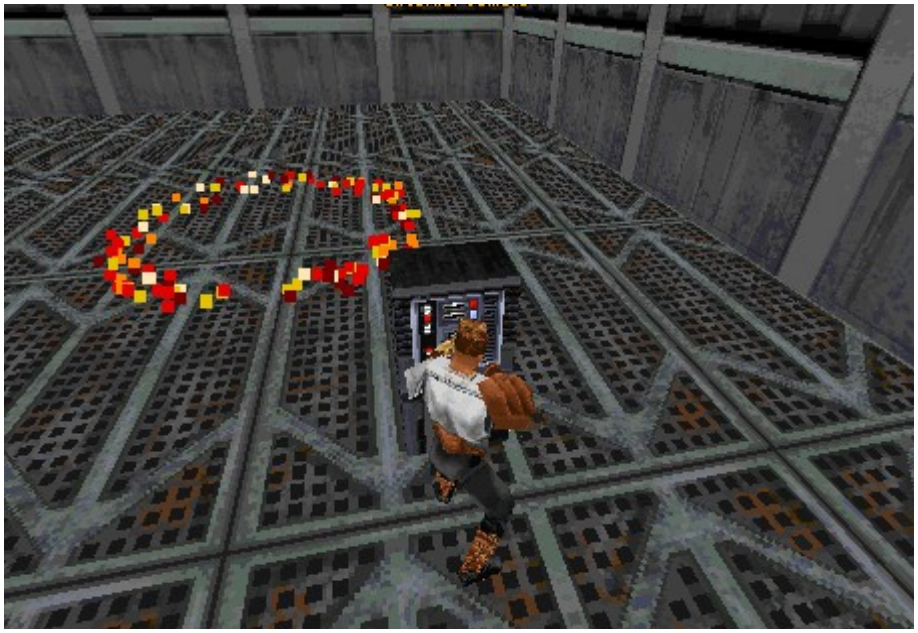


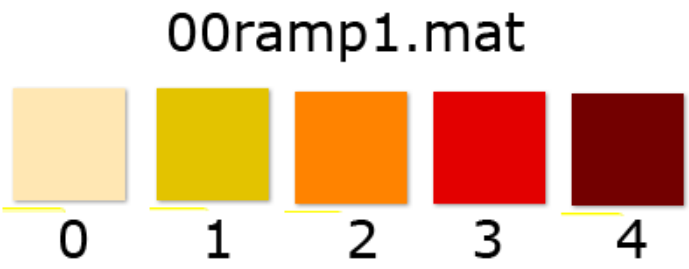
Illustration 34: Random start cel flag 0x04

Sure enough we get a random pattern of all the colors we expect to see in the sparks effect.

The unfortunate part about JED/ZED is, for whatever reason, it does not show *00ramp1.mat* to be multi-cel. Whacking together a quick script...

Illustration 35: cel counts for various particle effects

...we can see that actually an overwhelming majority of these materials used in particle effects are in fact multi-cel.



Drawing 26: sample multi-cel material used for sparks

As it stands, our effect results in particles being randomly assigned a cel but it doesn't change throughout the duration of the effect.

We have a differing explanation for particle flag 0x20: JKSpecs indicates this means “Flipped” whereas DataMaster indicates this means particles will cycle through cels.

Illustration 36: particle effect with 0x20 typeflag

It seems DataMaster was correct. We still have the 0x04 random starting cel flag enabled, so all the particles started out as different colors, but slowly **over time they cycled toward the last cel in the material**. What's interesting to note here is they did not loop back to the first cel or anything; they just stuck at the last color.

Reducing the *timer* attribute of the effect made the cel animation speed increase. It would appear as though it is intended such that all **particles will have made the transition to the final cel in the material as the same time as the timer expires**. Further, combining this effect with the 0x08 fade out flag, all particles still cycled to final color within the *timer* duration, then they started dissipating according to the *rate* parameter.

Expand outwards

The 0x01 flag causes particles to fly out from the centerpoint.

Illustration 37: Ring of particles expanding

The direction seems to always be from the particle's spawn point away from the central thing position (which is 0,0,0 in respect to the particle cloud).

The speed seems to be controlled by *maxthrust*. It should be noted that there does not seem to exist any template attributes like *minthrust* or *thrust*, nor do there seem to be any other parameters that affect a variability in the expansion rate.

Unknowns

Let's also take a look at the 0x10 flag: JKSpecs labels this as “Emits light” but DataMaster says it is unknown but suspected to be involved with cel animation. Since we're on that topic already and DataMaster tends to have the correct hunches, let's see what happens.

Toggling the 0x10 flag on and off was tested under a number of circumstances in combination with various other flags and parameters.

At this stage, no discernable difference was found. Also, there seems to be no obvious rhyme or reason to why some particle templates specify 0x10 and others don't.

Regarding light: the light level of the test chamber was brought very low and it actually appeared as though *all* particle effects appeared fully bright and *none* of them cast any dynamic light. This may be due more from the fact that the palette entries specified by the materials may point to self-illuminated colors (although this has not been checked).

Let's also take a look at the 0x02 flag: JKSpecs says that this one is for cel animation, but DataMaster patently states it is unknown.

Similarly to the 0x10 flag, no discernible effect was noticed under a few various combinations of flags and parameters.

Logic

Aside from some of the more specialized subsections such as Physics are another class of services that the engine provides. The Logic section will contain miscellaneous aspects that do not fit under a major system.

Sight

The concept of "sight" goes beyond simple visibility determination for the sake of rendering. AI characters have the ability to "see" and there is a slew of COG verbs that cover line-of-sight and thing-in-view functions to augment AI and provide a targeting framework for various force powers.

We will explore several areas in hopes to not only maximize the compatibility/recreation efforts but also to identify schemes to optimize these routines.

HasLOS

The COG verb "HasLOS" (has line-of-sight) has a fairly weak description according to DataMaster:

```
Tests to see if a target is in view of a look_thing and returns a
boolean value. The look_thing does not have to be facing the target for
HasLOS() to return true. Syntax:

boolean=HasLOS(look_thing, target);
```

Unfortunately, it does not indicate what types of things/surfaces can interrupt the line-of-sight, nor what kind of properties that thing/surface must have in order to be considered obstructing visibility.

A basic implementation in Smith performs a "raycast" using a line segment from the source to target, and if any "solid" thing/surface is hit rather than the target then the LOS check is deemed to fail. The "raycast" operation is relatively expensive in its current implementation; even though optimized using thing and sector BVHs.

In most circumstances the performance impact is completely satisfactory, however it has been noted in some very aggressive COGs that utilize HasLOS in a fast pulse running in many concurrent instances is a major performance problem. Of course, the same level runs just fine in JK. A particular example is in Edward's "The Rainbow Factory" level, with the script class_swinglamp.cog.

FINDINGS

This section consolidates the preceding analysis results into final rules and formulas.

PHYSICS

Drag

Dynamic Drag

When a thing is attached to a surface, DragCoefficient is surfdrag.

When a thing is not attached to a surface, DragCoefficient is airdrag.

Dynamic drag is applied to a thing as an acceleration defined by:

```
Acceleration += -Velocity * DragCoefficient;
```

Static Drag

When a thing is attached to a surface, has no acceleration and the magnitude of its velocity is at or below its staticdrag threshold then the velocity is set to 0.