Automatic Generation of Environmental Reverb Data

A Method for Selection and Blending of Reverb Presets Using Environmental Scanning Techniques

James Boer

Overview

As videogames grow ever-larger, often with huge, expansive worlds, we developers must look for new ways to improve our tools and technology to deal with the increased workload imposed by the sheer breadth of these virtual environments. This challenge applies to audio designers and the application of environmental reverb settings across the game world. Traditional methods often involve laboriously hand-marking regions in the world with appropriate-sounding reverb presets. Recently, more automated or dynamic methods of scanning the world and using real-time parameters to control reverb have been used with some success.

In this article, we'll discuss a method of using environmental scanning techniques to create a unique signature of an existing space's geometry and surface characteristics, and of using that signature to select a ratio of two preset reverb settings from an arbitrary set that best matches the aural characteristics of any given position in the world. The workflow for such a system is similar to the traditional method of creating reverb presets for specific environments, such as an open plain, a dense forest, a rocky canyon, a large cave, or a narrow stone hallway. However, instead of covering the entire world with these presets regions, we are only required to place a few key presets as "training" values.

We can then use an algorithm that compares these training presets to physical characteristics of the current environment, and determines the top two candidates among the reverb presets, plus a blending ratio. By using reverb presets in this way, we gain the advantage of using known-good starting points for our reverb, using hand-crafted presets created by audio designers. This also means that we can use convolution reverbs with this method, as they often don't lend themselves to dynamic parameter adjustments as easily as with algorithmic reverbs. By mixing the two closest matching presets in designated ratios, we gain the ability to generate location-specific reverb blends that more precisely match the individual characteristics of local terrain features, while still requiring relatively few hand-tuned reverb presets.

Defining an Environmental Region using a Raycast-based Scanner

At the heart of this technique is a straight-forward raycast-based scanner. The only requirement for the scanner is that it can detect collision geometry in the world and retrieve appropriate location and surface information from the collision. We then use these two key pieces of information, ***spatial data*** and ***surface types***, to build a profile of this location that uniquely characterizes its aural properties. This allows us to compare any two locations in the world and determine their differences, all reduced to a single normalized scalar value (0-1).

Analyzing Spatial Data

Let's examine a hypothetical environment to see how this might work. Imagine a stone corridor, which is perhaps a not-too-uncommon environment for a videogame.

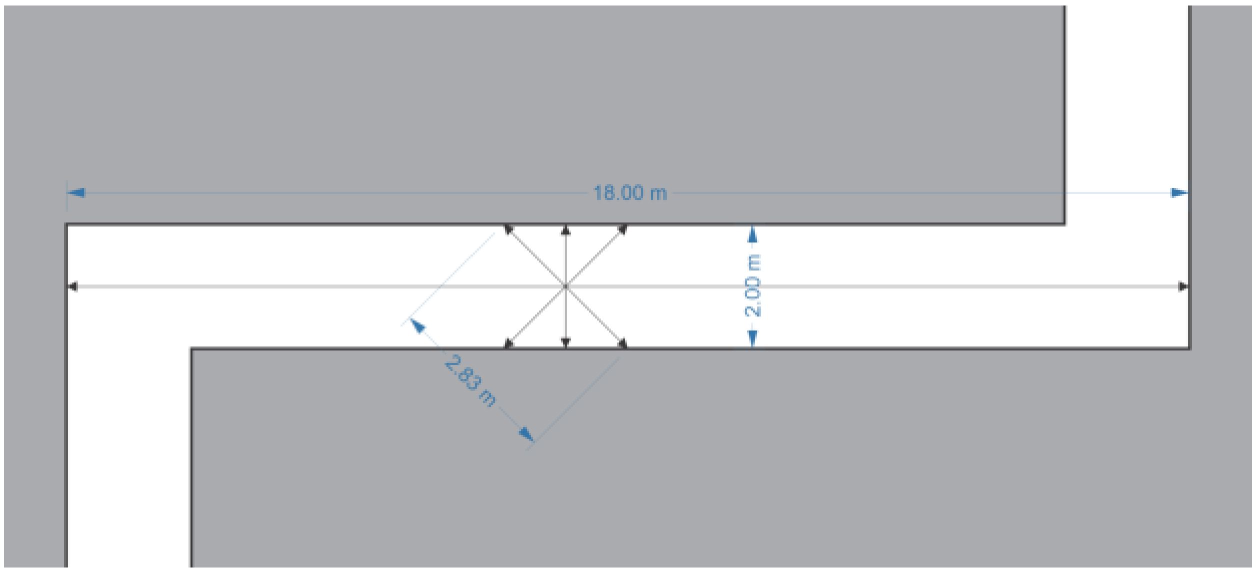


Figure 1 – Overhead view of hallway

In Figure 1, we show an overhead eight-way scan of a two-meter wide hallway. Naturally, in practice, we would likely be scanning in three dimensions and with greater resolution, but we'll keep things simple for the purpose of this demonstration.

The one-way scan distances from origin to collision could be somewhat useful in determining spatial characteristics, but there is a problem when using those values. The individual lengths will change drastically as you move around the room, leading to inconsistent results. Instead, a much more useful way to use these values is to combine the distances of the two scans in opposite directions. These values will not significantly change no matter where in the local area you move, because you are now measuring the dimensions of the room itself, not just relative distances from a given point.

So we now have a combined set of four two-way scan values measuring 2.00m, 2.83m, 2.83m, and 18m. Let's turn this into a structure that will be more useful for comparative analysis.

We'll first sort these values into distance-limited bins. Our distance values might look something like this:

```
const std::array<float, 11> DistanceBinValues =
{
    2.0f,
    4.0f,
    8.0f,
    16.0f,
    32.0f,
    64.0f,
    128.0f,
    256.0f,
    512.0f,
    1024.0f,
    std::numeric_limits<float>::max()
};
```

We're using power of two values to separate distances into bins ranging from a typical hallway size to over a kilometer away. Each value represents the maximum distance allowed in a given bin. The final bin represents raycasts that don't collide with any geometry, such as rays cast into the sky, or outward from a mountaintop.

We'll now create a set of bins for each maximum range value using a simple array of floats. Here's how we can do that:

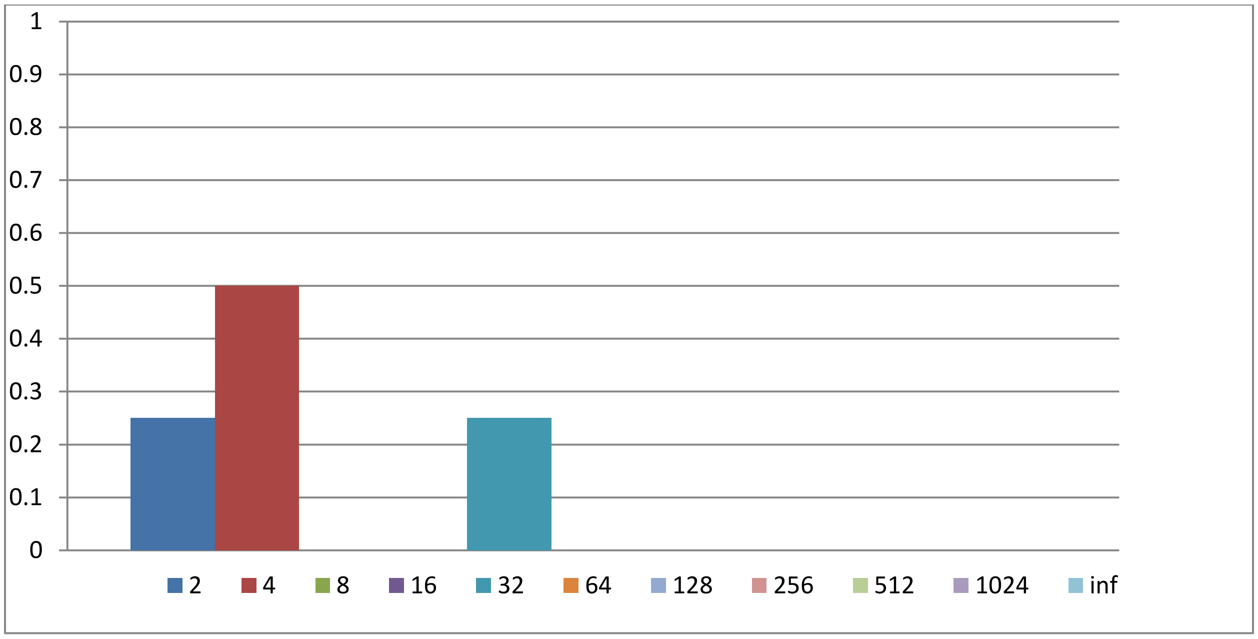
```
using DistArray = std::array<float, DistanceBinValues.size()>;

DistArray CreateHistogramFromDistances(std::vector<float> distances)
{
    DistArray histogram;
    for (auto d : distances)
    {
        for (size_t i = 0; i < DistanceBinValues.size(); ++i)
        {
            if (d <= DistanceBinValues[i])
                histogram[i] += 1.0f;
        }
    }
    NormalizeHistogram(histogram);
    return histogram;
}
```

We see at the end of this function that the histogram is normalized. That is, the total value of all bins should add up to one. That's a simple matter of adding up the values in all bins, then dividing each bin by that total.

```
template <class T>
void NormalizeHistogram(T & h)
{
    float total = 0.0f;
    for (auto v : h)
        total += v;
    for (auto & v : h)
        v /= total;
}
```

The resulting histogram should remain relatively consistent no matter where we move in our sample hallway. If we turn the hallway 45 or 90 degrees, the measurements will remain precisely the same. Even at other arbitrary angles, the values will still look reasonably similar. And of course, if we scan with more precision, say, using 32 horizontal raycasts instead of just 8, the consistency of the data will be improved still further.

Plotted out graphically, the data looks like this:



This distinctive shape, in the form of a distance-sorted histogram, is what allows us to find similar geometric patterns though the use of a histogram-matching algorithm, which we'll describe in more detail a bit later.

Let's measure another location in our hypothetical hallway, this time at a corner location, which should give us a slightly different histogram.

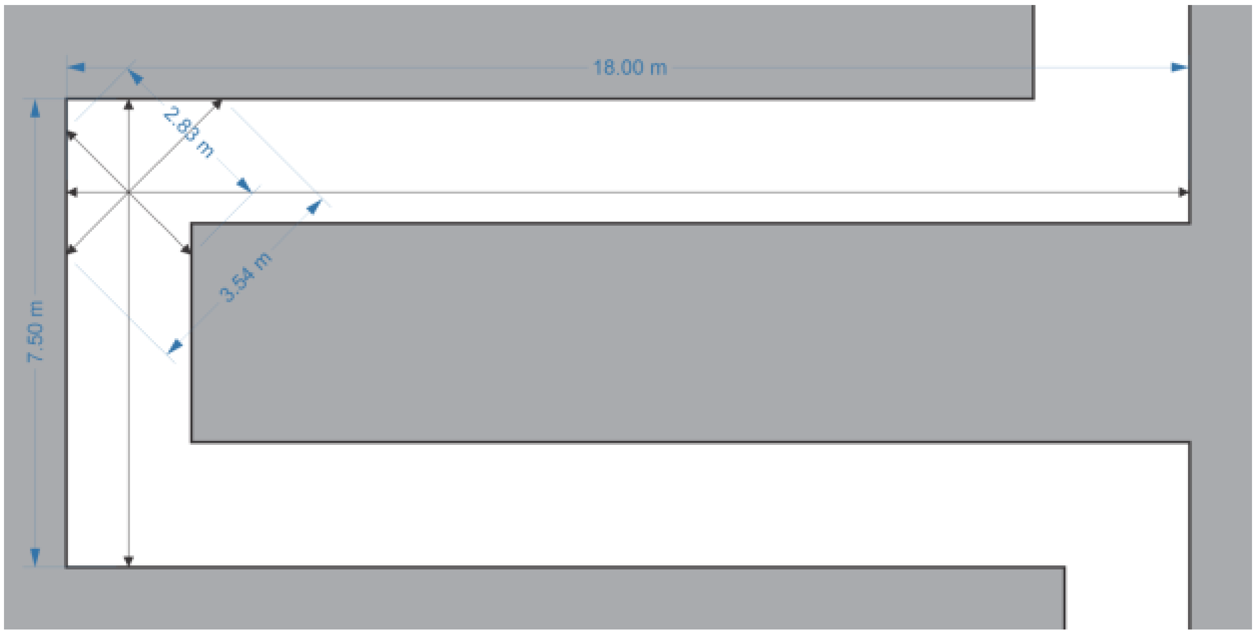
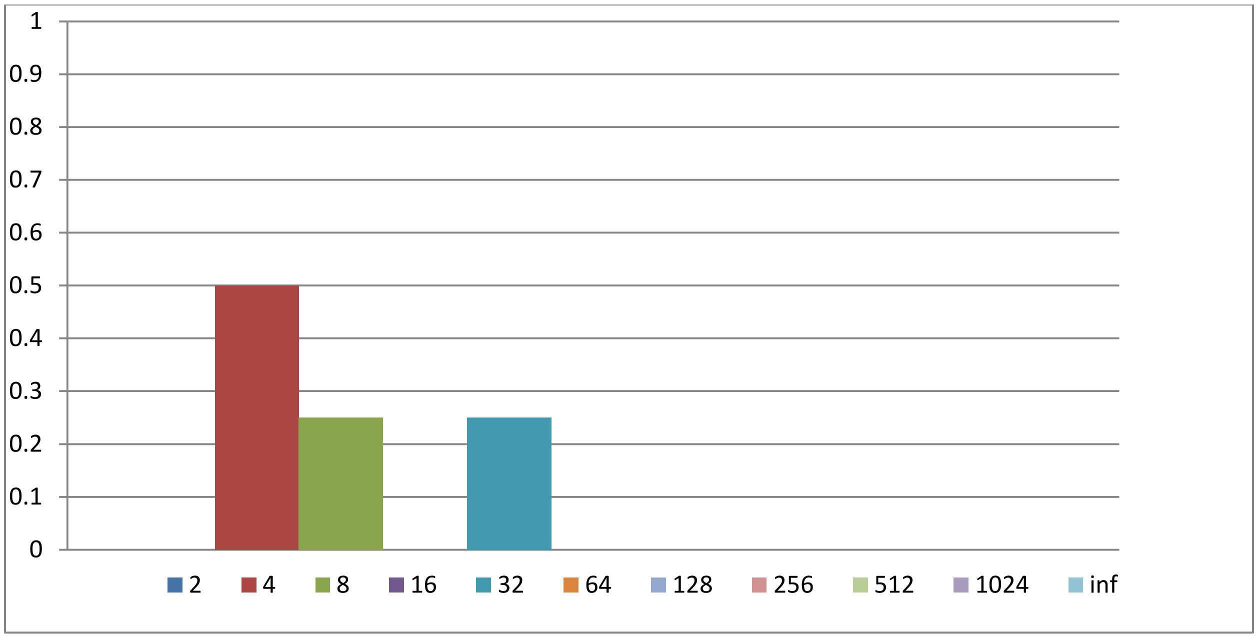


Figure 2 – A different scan location

We can immediately see that the scan measurements are a bit different. Here's what the second histogram looks like:



It looks fairly similar to the first histogram, at least visually.

Now let's explore how we can compare these two histograms, and calculate the difference between them. There are many such algorithms, but we'll base our solution on the ***Earth Mover's Distance (EMD)***[i] algorithm. It essentially computes the "work" required to transform one histogram into the shape of another, as though the values of the histogram were physical objects, such as piles of earth.

EMD can be expensive to calculate in the general case, but is efficiency computed in O(n) time for simple structures like histograms. A differential value is computed and carried over as the algorithm calculates the distances between successive points of the two histograms. This algorithm takes advantage of the fact that the two histograms are of equal size and are normalized.

Now let's examine our histogram comparison algorithm in detail. The function will return a value of 0 for an identical match, while 1 represents histograms that are as dissimilar as it is possible to be (this is defined as a value of 1 in opposite bins for the two histograms). Here is the code:

```
template <class T>
float GetHistogramDifference(const T & a, const T & b)
{
    assert(a.size() == b.size());

    // Track difference between histograms a and b
    float difference = 0.0f;

    // Accumulate excess values when equalizing histogram bins
    float accumulation = 0.0f;

    // Calculates cost of moving a full value (1.0) between adjacent bins,
    // based on the total histogram size.
    const float workScale = 1.0f / static_cast<float>(a.size() - 1);
    for (size_t i = 0; i < a.size(); ++i)
    {
        // Calculate a delta between the two histograms
        float delta = std::abs(a[i] - b[i]);

        // Determine the minimum value between the delta and our current
        // accumulation.  We can reduce our current accumulation and local
        // delta by this amount and no more.
        float reduction = std::min(delta, accumulation);
        accumulation -= reduction;
        delta -= reduction;

        // Store leftover delta value as accumulation for future bins to use
        accumulation += delta;

        // Add the difference from the last bin as the current accumulation
        // times the workScale
        difference += (accumulation * workScale);
    }
    return std::min(difference, 1.0f);
}
```

This function calculates a difference of 0.05, or 5%, between our two sample histograms. That's a fairly close match, so the same reverb preset chosen for the hallway is likely to be chosen for the corner as well.

Analyzing Surface Data

Geometric shape is not the only significant factor in determining the characteristics of a particular environment's reverb. The materials that make up these shapes also play an important role, mostly due to what we'll refer to as the ***absorption coefficient***[ii]. This refers to the amount of sound energy that is absorbed by the physical characteristics of the material. Hard, flat materials such as ice, steel, and stone have a very low absorption coefficient, while materials like snow, carpet, and foliage have a very high absorption coefficient.

While diffusion is a different physical property, determined by the shape of a surface instead of its physical characteristics, the results are still somewhat related, in that it will tend to affect the aural characteristics of a room. Rough, uneven values will tend to diffuse sounds more readily than smooth, regular surfaces. Thus, the relatively smooth walls of cut stone will reflect sounds more regularly and produce harsher echoes than the irregular shape of a cave's rock wall, even though the materials may be the same. You may wish to account for this by assigning slightly higher absorption values to surfaces with greater diffusion properties.

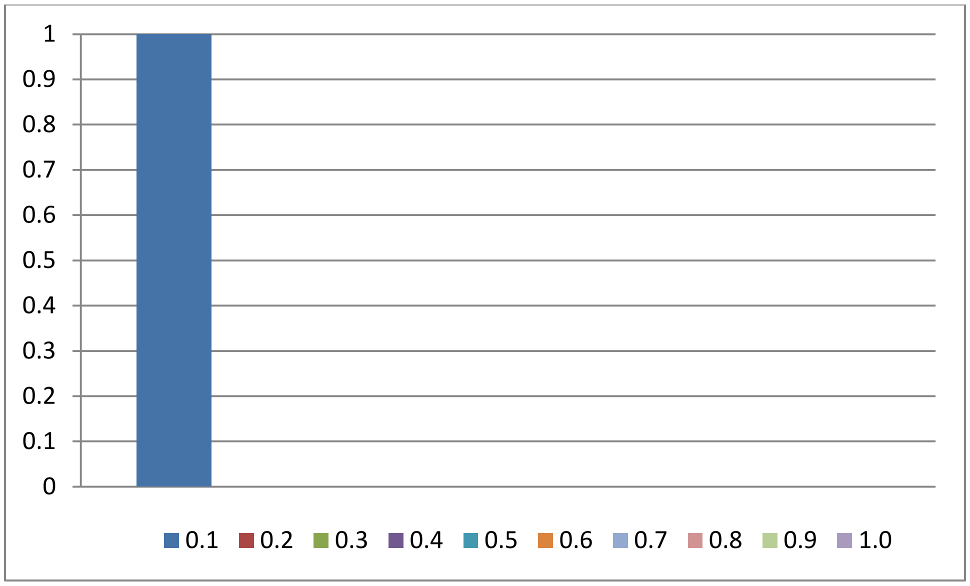
In theory, we might be able to automatically generate some additional diffusion statistics by measuring and comparing the angles of all the adjacent surfaces from our raycast collisions relative to each other, but we'll leave that for future consideration.

There are many reference tables available that describe the absorption coefficients of various materials. Technically speaking, these coefficients tend to vary across different frequencies, but for our purposes, it's sufficient to reduce this to a single representative value. Listed below are some such coefficients, derived from real-world tables of materials and tweaked for application in videogames:

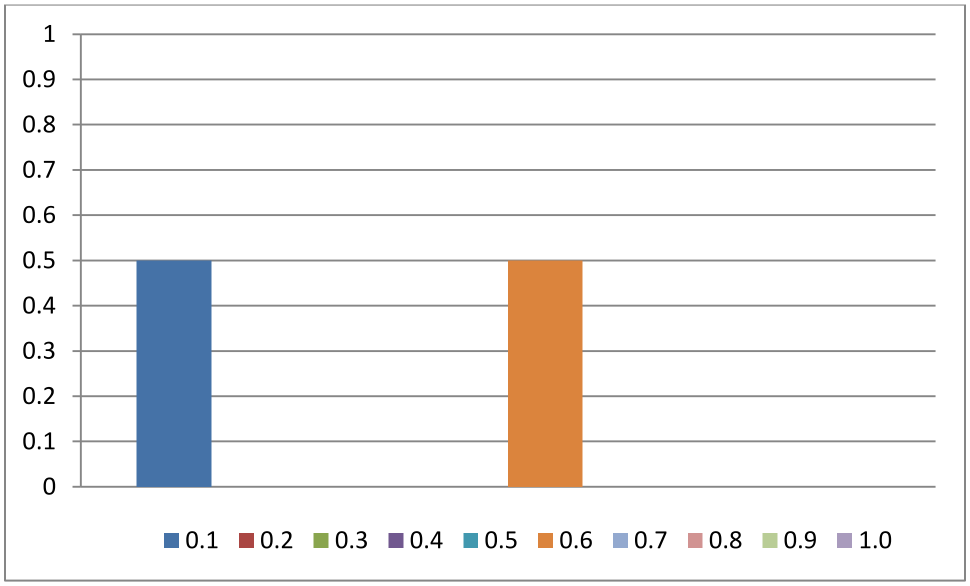
Material Name	Absorption Coefficient
asphalt	0.60
body / flesh	0.70
brick	0.35
carpet / cloth / fabric	0.65
ceramic / marble	0.01
composite / plastic	0.30
concrete	0.05
cut stone	0.10
dirt / soil	0.60
foliage / grass / moss / plant / vegetation	0.75
glass	0.10
gravel	0.40
ice	0.20
iron / metal / steel	0.10
rock	0.15
sand	0.55
snow	0.80
water	0.01
wood	0.25

In most game engines, a raycast will, on collision, return a material type of some sort so that a game can react appropriately, such as playing a particular footstep sound or emitting a particular visual effect. You will need to use this information to look up an appropriate absorption coefficient. When a raycast hits nothing, the absorption coefficient should be interpreted as 1.0, since there is nothing to reflect sounds from that direction.

We will gather this information into a histogram as well, with coefficient values sorted into bins by value. Let's divide these returned absorption coefficient values into ten discrete bins. A stone hallway with no other features will look like this:



That's a pretty distinctive histogram, so such a well-defined shape will tend to match with other similar, high-coverage / high-reflection environments. Let's assume for a moment that our theoretical stone hallway has its walls partly covered by massive tapestries (roughly 50% coverage). Now the histogram may look more like this:



This is rather a different shape, so has a better chance of matching a different preset, probably one with a less pronounced effect due to the increased absorption characteristics of the environment, although keep in mind this is entirely dependent on having set up such a preset in an appropriate environment. Our algorithm calculates the difference between these two histograms to be 0.25, or 25%.

Combined Analysis

We now have two histograms (distance and absorption coefficient) we can use as a unique "signature" for a given area's environmental audio characteristics, as defined by both shape and materials. Now we must also normalize the two difference calculation as well. Assuming we wish to give both histograms an equal weight, this is as simple as adding both difference values together and dividing by two, since they're also both normalized values.

```
float GetSignatureDifference(const SignatureData & a, const SignatureData & b)
{
    float distDiff = GetHistogramDifference(a.distances, b.distances);
    float absCoeffDiff = GetHistogramDifference(a.absorbCoeffs, b.absorbCoeffs);
    return (distDiff + absCoeffDiff) / 2.0f;
}
```

You may also wish to gather additional information or statistics, and use them in the signature data. For example, you could perform additional detection of vegetation which may be excluded from normal raycast hit detection, or factor in weather effects like fog, rain, or snow. Naturally, this will depend on engine-specific techniques to a significant degree.

Whatever additional information you analyze, you should ensure that the comparison function returns a normalized value. This will make it simpler to add additional information in the future without unduly impacting the rest of your code.

Manual Override Zones

It is important to keep in mind that automated systems rarely produce data that's 100% perfect. Your audio designers should have a method for manually overriding the system to tweak a particular area by hand. It's highly recommended to add an option to your reverb containers that allow them to simply force a preset to a particular value in that area at runtime.

Scanning

We need to implement a scanner to extract environmental information. The same scanning technique should be used for both generating signature data for the preset reverbs, as well as for generating data used to compare against those signature values. At a basic level, this simply involves casting a number of rays outward from a given position, then aggregating and analyzing the results.

There are two fundamentally different approaches when implement this type of scanning algorithm, and both have their own benefits and drawbacks.

Export-time preprocessing involves scanning the entire world during authoring or export time, then sorting and storing the data for real-time lookup. It is extremely run-time efficient, but less flexible than real-time scanning.

Real-time scanning is a great option if you don't want to bother with pre-processing and storing data, but it tends to lose a bit of fidelity compared to pre-processing and imposes a bit more run-time overhead. In compensation, this method more easily deals with dynamic geometry of any sort.

Preset Scanning

Before we discuss preprocessing vs real-time, both techniques will have one aspect in common. This involves the audio team placing a select number of preset reverb regions in the world. Unlike with typical reverb coverage, it's best to create a relatively small region that you can consider to be a "prototypical" example of a particular reverb preset. For instance, if you want to have a "stone corridor" preset, you'd place a small region right in the middle of the most typical example of a stone corridor in your game environment, and design the reverb specifically for that corridor.

The matching algorithm will check against these specific histogram values when trying to determine how closely the current environment matches the "ideal" of a stone corridor – essentially, the signature values associated with that preset region. We've seen earlier how slight variations in geometry and material will only produce a small differential value, while greater variations in sizes or material will produce larger differences on comparison.

Preset Signature Reduction

When we store our preset training values, it will be necessary to cap the maximum number of signature values per preset region. This is important, since checking for the closest signature match is an O(n) algorithm.   Depending on the size and density of the preset scan regions, there may be many thousands of signatures to preprocess. Many of these signatures may be a very close match to other signatures, and so are largely redundant.

Let's examine how we might want to cull our signatures to a more manageable number. We'll use two algorithms for culling. The first of these algorithms checks the difference between every two signatures, and removes any that fall below a specified threshold. This helps to reduce redundant signatures that have only minimal or no differences at all. Here is the function to do that:

```
static std::vector<SignatureData> CullSignaturesByDifference(
    const std::vector<SignatureData> & signatures, const float diff)
{
    // Copy signatures into intermediate structure for diff-based culling
    std::vector<SignatureData> s = signatures;

    // Mark all signatures within diff threshold of
    // another signature as invalid
    for (size_t j = 0; j < s.size(); ++j)
    {
        if (s[j].presetIndex == InvalidIndex)
            continue;
        for (size_t i = j; i < s.size(); ++i)
        {
            if (s[i].presetIndex == InvalidIndex || i == j)
                continue;
            if (GetSignatureDifference(s[i], s[j]) < diff)
                s[i].presetIndex = InvalidIndex;
        }
    }

    // Find and erase any invalid presets from the vector s
    s.erase(std::remove_if(s.begin(), s.end(), [&] (const auto & e) {
        return e.presetIndex == InvalidIndex;
    }), s.end());

    // Return vector of culled signatures
    return s;
}
```

The second algorithm reduces the number of signatures to a specified limit. Again, the distances between all signatures are calculated. This information is stored and sorted, and then used to find the specified number of signatures that have the greatest separation between them. This should help to provide the greatest variety of signatures available in the given set. Here's what that function looks like:

```
static std::vector<SignatureData> ReduceSignaturesBySize(
    const std::vector<SignatureData> & signatures, const size_t size)
{
    // Copy signatures into intermediate structure for size-based culling
    const auto & s = signatures;

    // Calculate all differences and store index and diff value in a vector
    std::vector<std::pair<size_t, float>> diffs;
    for (size_t j = 0; j < s.size(); ++j)
    {
        for (size_t i = j; i < s.size(); ++i)
        {
            if (i == j)
                continue;
            float delta = GetSignatureDifference(s[i], s[j]);
            diffs.emplace_back(i, delta);
        }
    }

    // Sort the diff vector from largest to smallest diff.
    std::sort(diffs.begin(), diffs.end(), [] (const auto & a, const auto & b)
    {
        return a.second > b.second;
    });

    // Copy the top unique presets into sigs
    std::vector<SignatureData> sigs;
    sigs.reserve(size);
    std::set<size_t> indices;
    size_t index = 0;
    while (sigs.size() < size)
    {
        if (indices.find(diffs[index].first) == indices.end())
        {
            sigs.push_back(s[diffs[index].first]);
            indices.insert(diffs[index].first);
        }
        ++index;
    }

    return sigs;
}
```

We'll now combine these two algorithms into a single signature reduction function. First we'll reduce the entire set of signatures by a differential threshold. We'll then organize the signatures by preset index. For each preset, we'll then reduce only that set to the specified maximum size. Here's the function that puts this all together:

```
std::vector<SignatureData> ReduceSignatures(const std::vector<SignatureData> &
signatures)
{
    // Adjustable constants
    const float MinViableDistance = 0.02f;
    const size_t MaxSignaturesPerPreset = 5;

    // Copy signatures into intermediate structure for initial distance-based culling
    std::vector<SignatureData> sigs = ReduceSignaturesByDifference(
        signatures, MinViableDistance);

    // Organize signatures by preset index for additional processing,
    // then clear sigs, which we'll reuse later.
    std::unordered_map<size_t, std::vector<SignatureData>> sigMap;
    for (const auto & s : sigs)
        sigMap[s.presetIndex].push_back(s);
    sigs.clear();

    // Reduce number of signatures for each preset index to no more
    // than MaxSignaturesPerPreset
    for (auto & sp : sigMap)
    {
        auto & s = sp.second;

        // Reduce numbers if the vector size is greater than the threshold
        if (s.size() > MaxSignaturesPerPreset)
            s = ReduceSignaturesBySize(s, MaxSignaturesPerPreset);
        // Copy remaining elements to final output vector
        sigs.insert(sigs.end(), s.begin(), s.end());
    }

    return sigs;
}
```

You may notice that these algorithms run in O(n^2) time. This can't be helped, as we're performing a exponential algorithm when we compare every signature to every other signature. For this reason, you may wish to put some hard limits on the number of signatures initially collected. For instance, you could enforce a maximum size or maximum scan resolution for preset regions to avoid an excessive number of initial signatures.

Scan Techniques

When scanning in three dimensions, a basic technique for generating scans is to successively walk the vertical angle up/down from 0 to 90 degrees in x degree increments (e.g. 15 degrees) while rotating the overhead angle from 0 to 360 degrees in a radial pattern.

However, we'll want to scale the number of radial scans by the cosine of the elevation angle. If we apply this formula, then, for example, with an initial 32 radial scans at 0 degrees of elevation, we would scale number that by .707 at 45 degrees for 22 radial scans, .5 at 60 degrees for 16 radial scans, and so on. These vectors should be calculated at initialization time and stored as unit vectors for convenient access at runtime.

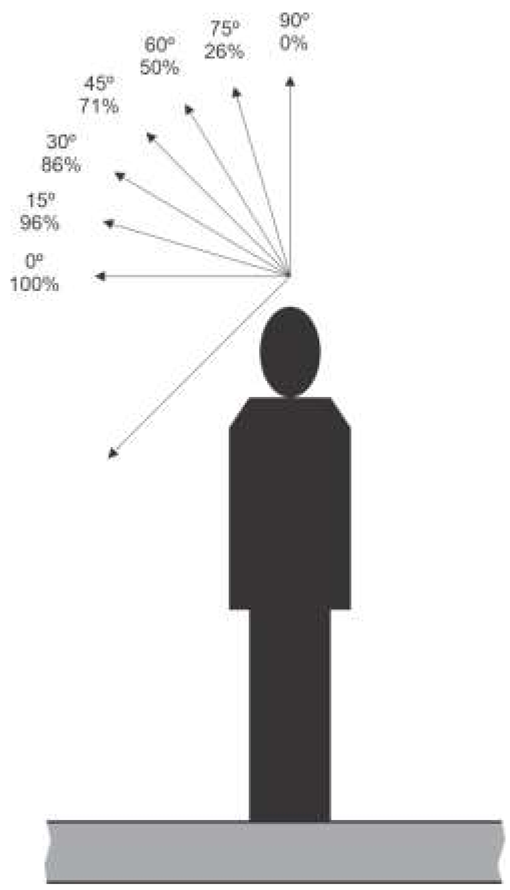


Figure 3 – Elevation-based scaling of horizontal scan numbers

While this gives a reasonable equidistant scan pattern, a more robust formula exists [iii] for calculating a spherical pattern, which involves wrapping a Fibonacci spiral around a unit sphere. Here is a function[iv] that demonstrates how to do this.

```
std::vector<Vec3> fibonacci_spiral_sphere(const int num_points)
{
    std::vector<Vec3> vectors;
    vectors.reserve(num_points);

    // golden ratio = 1.6180339887498948482
    const double gr = (sqrt(5.0) + 1.0) / 2.0;
    // golden angle = 2.39996322972865332
    const double ga = (2.0 - gr) * (2.0 * M_PI);

    for (size_t i = 1; i <= num_points; ++i)
    {
        const double lat = asin(-1.0 + 2.0 * double(i) / (num_points + 1));
        const double lon = ga * i;

        const double x = cos(lon) * cos(lat);
        const double y = sin(lon) * cos(lat);
        const double z = sin(lat);
        vectors.emplace_back(x, y, z);
    }
    return vectors;
}
```

For any given method of determining your scan vectors, you must find and store all reciprocal vectors' indices. This can be done independent of the generation algorithm by performing a dot product against all other vectors, and selecting the vector that is closest to a value of -1, which represents unit vectors in opposing directions. This information can then be stored in an array of reciprocal indices for fast runtime access. Your code to do this may look something like the following:

```
std::vector<Vec3> GetReciprocals(const std::vector<Vec3> & scanDirs)
{
    std::vector<Vec3> reciprocals;
    reciprocals.reserve(scanDirs.size());
    for (size_t j = 0; j < scanDirs.size(); ++j)
    {
        size_t bestIndex = 0;
        float bestDot = 1.0f;
        for (size_t i = 0; i < scanDirs.size(); ++i)
        {
            if (i == j)
                continue;
            float dot = DotProduct(scanDirs[i], scanDirs[j]);
            if (dot < bestDot)
            {
                bestDot = dot;
                bestIndex = i;
            }
        }
        reciprocals.push_back(scanDirs[bestIndex]);
    }
    return reciprocals;
}
```

One potential problem with omnidirectional raycasts is the tendency to over-represent materials close to the scan origin. For example, the materials the character is standing on or a nearby wall may be over-represented in the histogram.

We can see in Figure 3 below how rays cast close to any given surface will tend to exaggerate the characteristics of the nearby surface simple due to its close proximity. In this case, even though the majority of the room is covered in stone, five of the eight raycasts will instead report a carpeted surface. A naïve collection of surface values would assume the room's stone-to-carpet ratio is 3 / 5, which is clearly not representative of the room.

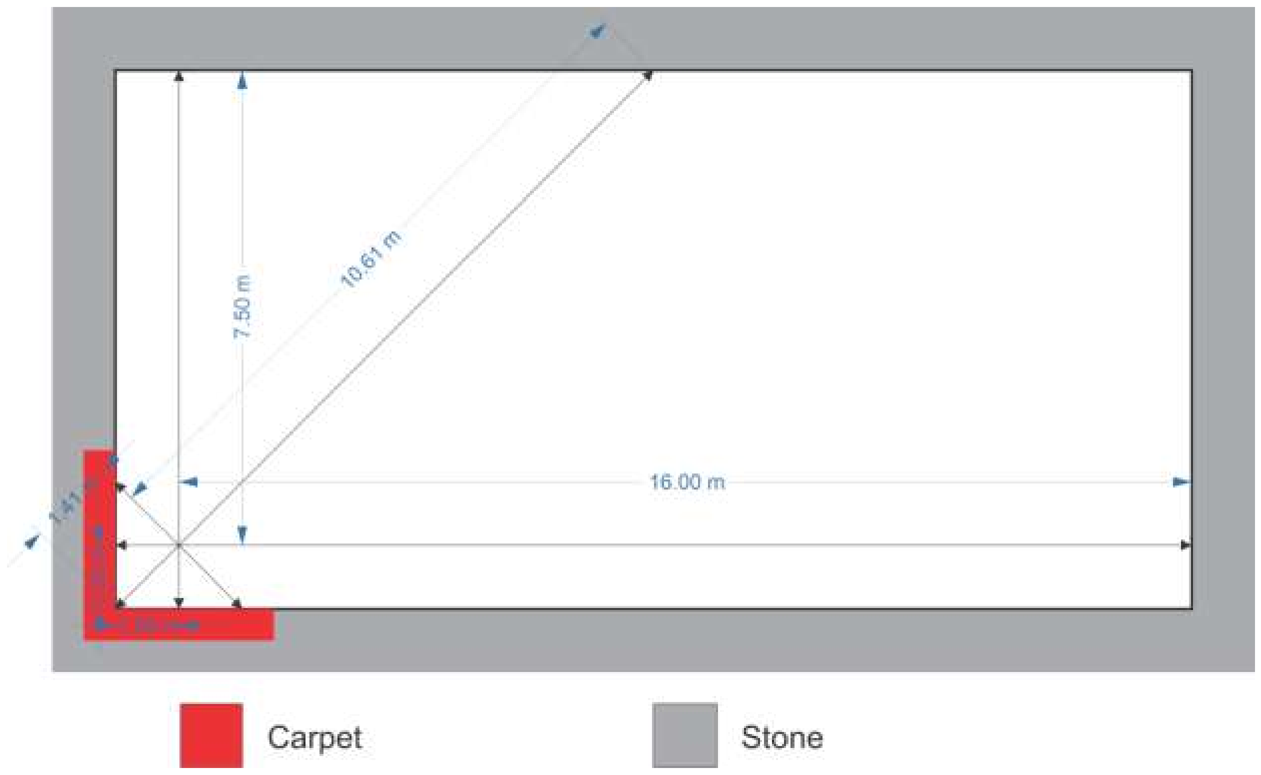


Figure 4 - Over-representation of materials in nearby geometry

We can compensate for this to some degree by weighting the surface absorption values by distance from the scan origin, so that nearby surfaces won't unduly affect the shape of the histogram. However, we won't be using the distance in a linear formula. Given the limited number of rays cast and the nature of the scanning pattern against geometric surfaces, it seems best to weight distant surfaces a bit more conservatively than that. Otherwise, a single ray that happens to hit different surface far in the distance could dramatically alter the histogram characteristics.

We can use a function like this to create a tweakable distance-based logarithmic formula:

```
const float DistWeightAdj = 1.6f;

float GetDistanceWeight(float distance)
{
    return std::pow(std::log(std::clamp(distance, 2.0f, 1024.0f)), DistWeightAdj);
}
```

Our baseline curve is created by the natural log function and we adjust that result by the power function, using `DistWeightAdj` to adjust the curve rate. We also clamp the distance input to work better with our functions and their viable ranges. Using this formula, we get the following weights at these distances:

Distance	Weight
2	0.556315
4	1.68643
8	3.22637
16	5.1123
32	7.30588
64	9.78054
128	12.5163
256	15.4976
512	18.7115
1024	22.1473

Plugging the scan results from our example from Figure 3 into our distance-based weighting formula, we now get a stone-to-carpet ratio of 4.35 / 1, which is a much better (if slightly conservative) representation of the actual room characteristics. Increasing the formula's curve constant will create a steeper curve, with more aggressive distance-based fix-ups, at the risk of distant scan results overly-affecting the histogram shape, as mentioned earlier.

Somewhat related to this issue, it is typically better for a character's scan to originate from their head or some high point on their body (perhaps even slightly above), rather than scanning from a character's midsection or feet, which will only exacerbate this problem.

Export-time Preprocessing

With export-time preprocessing, we can be much more liberal in our application of raycasts to profile the local environment. Being an "offline" method allows more accuracy with the scanning process, but does require additional run-time storage space. The benefits include very fine-grained environmental data auto-generation. Moreover, with a good spatial organizing data structure (octree, K-D tree, etc), lookup time for this data can be extremely efficient.

This means that it's also quite feasible to implement *per-source reverb*, since looking up environmental data for any given location in the map is a very lightweight operation. The downside is that preprocessing only works with static geometry. If you have a significant amount of dynamic geometry in your game that could affect reverb characteristics, you might wish to consider real-time scanning, or implement a hybrid scheme.

Preprocessed data can also benefit from additional methods of processing the raw data. For example, you can apply a smoothing algorithm across the entire data set to reduce the likelihood of any small, local scanning anomalies which may occur.

Real-time Scanning

Real-time scanning requires a bit more finesse, as raycasts can be somewhat expensive to perform even for modern machines. So it is unlikely you can perform as many raycasts as you might do during a preprocessing scan.

One potential solution is to perform these raycasts over time instead of simultaneously. Many in-game avatars do not move a significant distance in a short period of time, so even casting several rays per frame at 30-60fps means you can perform a fairly comprehensive scan in well under a second. This does introduce a small bit of lag in the application of reverb, but in practice, this is very hard to notice. For best results, you can stagger the raycasting pattern over time to ensure a more uniform construction of the histogram data. This can be done by storing a separate array of index values that are shuffled with a fixed-seed random function, which can then be used to order the raycast operations.

Another optimization which may be employed is to reduce the frequency of raycasts for sound emitters that slow down or stop moving altogether. This optimization, however, must be weighed against the possibility of dynamic changes to nearby terrain, so caution is warranted.

It may still be possible to implement per-source reverb using real-time scanning, but it will be important to build a more robust system that can throttle the number of raycasts per frame, since now every source emitter in the scene must continuously scan the environment. Other optimizations can be applied, such as lowering the scan rate for distant emitters, or for emitters that are not moving.

A particular problem for real-time scanning occurs when emitters suddenly appear in the world and are expected to emit sound immediately. Unlike with preprocessed data, in which it is efficient to retrieve information on demand, real-time scanning is not something that can done instantly. It is expected that the default scanner not only throttles the rate of raycasts, but also likely executes them asynchronously via a job system or background thread.

The first potential solution is to create a “fast” option for scan requests, where a larger number of scans are executed immediately in order to generate a faster result than normal. This may involve the creation of a synchronous API, although this should be done with caution. A second solution is to use hints from nearby environmental scans to create an initial “best guess” as to the likely reverb presets for the new location. The closer the existing emitter is to this new emitter, the more confident we can be of the result.

Note that for this reason, it is important to continue to update emitters’ reverb preset data, regardless of whether they are playing sounds or not, because it is typically impossible to predict when they may be expected to play a sound on demand.

Blended Preset Results

The final output of our location-based reverb query is two reverb presets and a blend ratio. If one of the presets is close enough to a 100% match of the current location, then the ratio should be a value of 1.0. Only one preset is valid in this circumstance.

Note: A helpful debug technique involves creating a HUD to display the reverb data for the current avatar’s location. This lets you continuously monitor results as you run around the virtual world, giving you a good idea if the algorithms are working as intended.

For pre-generation of reverb data, this lends itself to very efficient storage requirements: three bytes per node should be sufficient. It would be rare to have more than a dozen or two discrete reverb presets, and a byte is sufficiently granular for the ratio.   The reverb values stored should just be indices into the list of reverb presets stored in your map.

The advantages of returning a ratio of two presets instead of just a single preset reverb are considerable. Not only does this allow for smooth blends between distinct environments, but it works without having to modify real-time parameters of the reverbs – only the aux send levels per emitter need be adjusted.

The “blended” nature of these preset pairs also manifests in more subtle ways. Entire environments may be best represented as a blend between two preset signatures. For instance, a medium sized room may be represented as 50% “large room” and 50% “small room” presets, and will manifest as a natural blend between those presets. This works surprisingly well in practice, and creates a much more subtle application of reverb than could ever realistically be applied by hand.

We’ll now show some code demonstrating how you might calculate a structure containing two preset indices and a ratio value between them. The function input takes two index and two difference values representing the relative difference between each of these presets and the current location.

```
const size_t InvalidIndex = std::numeric_limits<size_t>::max();

struct PresetPair
```

```
{
    size_t presetIndexA = 0;
    size_t presetIndexB = InvalidIndex;
    float ratio = 1.0f;
};

PresetPair CalculatePresetPair(size_t indexA, float diffA, size_t indexB, float diffB)
{
    if (diffB < diffA)
    {
        std::swap(diffA, diffB);
        std::swap(indexA, indexB);
    }
    PresetPair pair;
    pair.presetIndexA = indexA;
    if (diffA > 0.05f || diffB < 0.4f)
    {
        pair.presetIndexB = indexB;
        float totalDiff = diffA + diffB;
        pair.ratio = (totalDiff - diffA) / totalDiff;
    }
    return pair;
}
```

Let's assume we've determined that index A and index B represent the two closest differential matches to our current location. In this hypothetical example, we'll also assume that index A has a difference of 15%, while Index B has a difference of 30%. Our function will calculate this as a ratio of 0.66.

Note that if the difference for index A is less than 5% and the difference for index B is greater than 40%, we return only a single valid reverb index A and a ratio of 1.0, since in this case the ratio would otherwise exceed 85% anyhow.

Reverb Bus Management

Using pairs of reverbs per emitter has some implications for the number of active reverb buses. If we plan to use a global reverb for all sounds, then we will likely only require three buses – two for the currently active presets and one for transitioning.

However, if we wish to use per-emitter reverb, then the number of reverb buses is bounded by the number of current emitters x 2, up to the maximum number of presets. Clearly, this may not be practical, especially for more expensive effects, such as convolution reverbs. Thus, we must manage the set of currently active reverb buses ourselves.

Let's assume we wish to set a maximum number of five simultaneous reverb buses, which seems to be a reasonable target for modern hardware. This gives our environment at least two discrete pairs of presets, plus one slot used exclusively for transitions. While this may seem like a low number to service all audio sources, in practice, it is rare to see too many distinct reverb presets within a nearby region. The most common scenario is a fairly uniform set of one or two presets, with another one or two presets used in nearby geographical regions.

Prioritization and Preset Selection

Since we may have more active reverb presets in use than we have available buses, we need to determine a priority scheme to ensure a reasonable method of reverb bus selection.

- Nearby emitters should have a higher priority than distant emitters
- Priority should be increased based on the number of emitters using a given preset

To calculate priority from emitter distance, I would recommend a reverse exponential decay function, like this:

```
float GetEmitterPriority(float distance)
{
    return std::pow(0.9f, distance);
}
```

At a distance of 0, this returns a value of 1. By 8 meters, it returns 0.43. At 32 meters, it returns 0.034. This steep curve ensures that nearby emitters are always given top priority. The number of emitters can just be an additive value.

When calculating priority, we should create a running total for each preset. For every emitter, we calculate a base priority using our distance-based priority formula. We multiply this base priority by the output ratio for our two presets, and then add those values to each preset's running total. For a preset ratio of 1.0, we have only a single valid preset, and the entire priority score is added to that preset's total priority.

After all the emitters are calculated, we can sort the collection of preset priorities. We now have a target to match our top four prioritized preset reverbs to. If the sorted collection matches our current set of bus slots, then we do nothing. If the list does not match, and our available transition slot is currently fading out, then we also do nothing. If the list does not match and we have a free transition slot, we then choose the lowest priority bus slot and begin fading it out using the transition slot, while at the same time fading the new highest priority missing preset in. A half second transition time should work well, but this is a matter of personal preference.

In this way, we ensure that the four closest and most-used presets are active, while less important presets are pushed out of the set. This is the essence of managing the auxiliary reverb buses in real-time. The concept is straightforward, but it's a bit tricky bit of code to get working perfectly. To ensure the system is working as intended, I would highly recommend you add a visualizer that displays the buses in order of priority, showing their names and current fade value in some graphical manner (such as the length of a bar).

Preset Assignment to Emitters

One aspect you must deal with is what to do about emitters that are lacking one or both presets from among the currently active buses. There are currently two options I can recommend.

The first option is to simply replace any missing effect with the highest priority bus effect. This may sound like an odd solution, but there is logic to it. The highest priority bus is very likely to be the preset used for the environment nearest to the listener. Even sounds playing outside this environment will, technically speaking, still have to pass into this environment, and therefore are still likely to be affected. The end result is typically subtle enough that an average player will not notice this discrepancy.

The second option, perhaps a bit more risky, is to replace the missing preset with the available bus preset that is the closest match, difference-wise. For instance, a “small padded room” reverb might be substituted for a “medium padded room” reverb if it is otherwise available at the moment. One way to mitigate the potential risk of an inappropriate match could be to reject matches beyond a minimum threshold distance. If no match is found, we would want to revert back to highest priority bus by default. I’ve not used this method in practice myself, but have theorized that it should be feasible, and may provide even superior results.

A third option, only available for emitters with one of their two presets available, is to mix the available preset effect at 100%, instead of the designated ratio.

Keep in mind that you should not switch reverb buses on an emitter abruptly. This means that you’ll need per-emitter faders as well to ensure that if you must fall back to an existing bus due to one of its presets being removed from the active set, you can smoothly transition to the new bus with a fade-in.

Now with this infrastructure in place it should be a straightforward matter to associate each emitter with a pair of reverb bus identifiers, along with wet send values.

Conclusion

The techniques described in this article are the current culmination of my theoretical and practical work in the field of automated generation of reverb data, dating back to my 2002 book **GAME AUDIO PROGRAMMING**. I implemented and validated many of those theoretical ideas in **GUILD WARS 2**, and later made some substantial algorithmic improvements in Relentless Studio's **CRUCIBLE**.

While writing this paper, and with the benefit of hindsight, I have suggested many additional improvements to issues I came across during these projects as well. As these proposed methods are based on a solid foundation of both theory and practical experience, I'm reasonably confident that those theoretical improvements will hold true in practice. The core tenet of the entire system, the method of matching audio regions' physical characteristics to key training presets using raycast-based scanning and a histogram differential algorithm, remains fundamentally unchanged since its last application in practice.

Naturally, one should always double-check one's assumptions, especially when those assumptions are not thoroughly tested by a concrete implementation with real code in a real project. Consider this a fair warning that you may still need to experiment and tweak values and methods to get the results you desire.

You may be wondering if there is a working example of this system, for which you can examine the source in detail. Unfortunately, such an undertaking is beyond my current means, so I've attempted to recreate the most critical algorithms with actual code, while describing the rest of the system in as much detail as is feasible. If you have any questions, please feel free to contact me at james.boer@gmail.com, and I'll do my best to answer.

---

[i] https://en.wikipedia.org/wiki/Earth_mover%27s_distance
[ii] https://en.wikipedia.org/wiki/Absorption_(acoustics)
[iii] https://bduvenhage.me/geometry/2019/07/31/generating-equidistant-vectors.html
[iv] https://github.com/bduvenhage/Bits-O-Cpp/blob/master/geomtry/main_3D_fibo.cpp