Overall Sampling Process

Given: A set of visual concepts C, corresponding areas of the stage A, number of threads n, a global set of results $R = \emptyset$.

Algorithm 1: Search Initialization

```
Function startSearch(C, A)
    S ← computeSchedule(C, A)
    for i ← 1 to n do
        runSearchInThread(S)
```

computeSchedule returns a Schedule, which is an array of Samplers.
See Samplers section for how getSampler works.
getPinnedDevices returns two sets of ids: intensity pinned lights, and color pinned lights. See Pinning section for details.

Algorithm 2: Schedule Computation

```
Function computeSchedule(C, A)
    S ← ∅
    (intensityPins, colorPins) ← getPinnedDevices()
    for i ← 1 to size(C) do
        S ← S ∪ getSampler(C[i], A[i], intensityPins, colorPins)
    S ← S ∪ getPinSampler(intensityPins, colorPins)
    return sort(S)
```

The samplers are sorted such that a sampler that is completely contained within the region of another sampler comes after the containing sampler. Since things are sampled in order of the samplers in this array, we want things that affect the overall picture to happen first. Also intensity samplers occur before color, as color is somewhat dependent on intensity samplers to work properly.

See Samplers section for details on how getSample works.
getScores returns a map of scores resulting from evaluating the samplers' scoring functions

Algorithm 3: Search Thread

```
Function runSearchInThread(S)
    failures ← 0
    initThreshold ← 20                    // tunable param - initial JND threshold
    thresholdDecayRate ← 0.1              // tunable param - how quickly the threshold loosens
    threshold ← initThreshold
    while not done do
        s ← getCurrentState()
        for S_i ∈ S do
            s ← S_i.getSample(s)
        if isDifferent(s, R, threshold) then
            s.scores ← getScores(s, S)
            R ← R ∪ s                     // see Display section for additional details (clustering)
        else
            failures += 1
            threshold ← max(initThreshold / ((failures * thresholdDecayRate) + 1), 0.1)
```

Note: s is initialized by $getCurrentState()$ which returns the current state of the lighting configuration. s is always reinitialized to the current state before applying the schedule.

Visual Objective Implementation Details

Objectives will always sample by unpinned light groups. What this means is that if two devices are both unpinned and part of the same light group, the objectives will make sure those two devices get the same value for whatever is being sampled.

Intensity

The brightness visual objective is a Gibbs Sampling process that returns a new lighting configuration based on the initial configuration of the lighting when the search was started. Note that at the end of the Gibbs sampling process, the intensities are normalized to account for how many pixels of the rendered image the light actually affects.

Parameters determined from image: $mean$ - average brightness of all lights, $brightMean$ - brigthness of key lights, k - number of key lights. (The sampler accepts variance as well, but those are set to 10% of the full range as default for the $brightMean$ and 5% for the $mean$ and not free parameters at the moment.)

Parameters set when the search starts: $localDevices$ - set of devices affected by the objective, $pinnedDevices$ - set of devices that are pinned.

Intensity implements getSample as written in Algorithm 4 where s is the current state of the stage.

$computeSensitivity()$ returns an array of the sensitivity values for each affected system. Sensitivity is computed by rendering the lights in each system by themselves at 50% and 51%, taking the difference between those two images, calculating the average difference in brightness, and multiplying that by 100. In the actual code this is a preprocess operation as we can compute local systems before the sampler runs.

$getAllSystems()$ returns a set of devices in the entire lighting rig such that each set contains lights in the same light group.

$GibbsSamplingGaussianMixturePrior()$ is the function that performs the Gibbs sampling process. Psuedocode found in Algorithm 5.

Color

Parameters determined from image: $colors$ - map of color to relative frequency of that color. Can also be changed by user.

Psuedocode found in Algorithm 6.

$ClosureOverUnevenBuckets()$ Code listed in Algorithm 7.

Algorithm 4: Intensity Sampler

```
Function getSample(s)
    localSystems = []
    globalSystems = getAllSystems()
    constraints = []
    results = []
    /* Compute local systems                                                    */

    for system ∈ globalSystems do
        local = {}
        avg = 0
        affectedDevices = 0
        for device ∈ system do
            if device ∈ localDevices then
                affectedDevices+ = 1
                if device ∉ pinnedDevices then
                    local+ = device
            avg+ = device.intensity
        if affectedDevices > 0 then
            if local.size > 0 then
                constraints.append(FREE)
            else
                constraints.append(PINNED)
        else
            /* If there are no devices relevant to the system in this sampler, it's neither pinned or
               unpinned and is skipped                                          */
            continue
        results.append(avg / system.size)
        localSystems.append(local)
    sensitivities ← computeSensitivity(localSystems)
    intensities ←
    GibbsSamplingGaussianMixturePrior(results, constraints, sensitivities, size(results), k, brightMean, mean)
    /* Assign Intensities to System                                             */

    for i = 0 → localSystems.size do
        local = localSystems[i]
        for device ∈ local do
            deivce.intensity = intensities[i]
```

Algorithm 5: Gibbs Sampler

Function *GibbsSamplingGaussianMixturePrior(resultsIn, constraints, sensitivities, n, k, brightMean, mean, sigmaHigh = 0.05, sigmaLow = 0.05)*

$sampleMean = 0$
$sampleK = 0$
$nSampled = 0$
$results = resultsIn$
`/* For pinned systems, factor in unchangable intensities */`

for $i = 0 \rightarrow results.size$ **do**
  if $constraints[i] == PINNED$ **then**
    $currentSample = results[i]$
    if $currentSample > brightMean - mean$ **then**
      $sampleK+ = 1$
  $nSampled+ = 1$
  $sampleMean = ((nSampled - 1) * sampleMean + \frac{currentSample}{nSampled})$

$unconstrainedSamples = [0] * (n - nSampled)$
for $i = 0 \rightarrow unconstrainedSamples.size$ **do**
  if $constraints[i] == FREE$ **then**
    $nLeft = n - nSampled$
    $targetK = max((k - sampleK), 0)$ `// Target number of directed lights`
    $targetWH = min(\frac{targetK}{nLeft}, 1)$ `// Target high mixture weight`
    $targetMA = \frac{n*mean - nSampled*sampleMean}{nLeft}$ `// Target mean for rest of lights`
    $targetML = \frac{nLeft*targetMA - targetK*brightMean}{nLeft - targetK}$ `// Target mean for low intensity gaussian`

    $currentSample = 0$
    $currentMixture = \mathtt{U}(0, 1)$

    if $sampleK \geq k$ **then**
      $currentMixture = 1$ `// Enough bright lights`
    else if $k - sampleK \geq nLeft$ **then**
      $currentMixture = 0$ `// Not enough bright lights`
    if $currentMixture < targetWH$ **then**
      $currentSample = N(brightMean, sigmaHigh)$
    else
      $currentSample = N(targetML, sigmaLow)$
    $currentSample = clamp(currentSample, 0, 1)$
    if $currentSample > brightMean - sigmaHigh$ **then**
      $sampleK+ = 1$
    $unconstrainedSamples[i] = currentSample$
    $nSampled+ = 1$
    $sampleMean = ((nSampled - 1) * sampleMean + \frac{currentSample}{nSampled})$
$shuffle(unconstrainedSamples)$
$j = 0$
for $i = 0 \rightarrow unconstrainedSamples.size$ **do**
  if $constraints[i] == FREE$ **then**
    $result[i] = unconstrainedSamples[j]$
    $j+ = 1$
`/* Adjust intensities according to sensitivity values */`
$invS = map(f : x \rightarrow \frac{1}{x}, sensitivities)$
$sumInvS = \sum invS$
$invS = map(f : x \rightarrow \frac{x}{sumInvS} * n, invS)$
$results = min(results \cdot invS, 1)$
return *results*

Algorithm 6: Color Sampler

```
Function getSample(s)
    intensity = []
    bins = [0] * colors.size
    localSystems = []
    globalSystems = getAllSystems()

    /* Compute local systems and fill in color buckets with pinned colors */
    for system ∈ globalSystems do
        localSystem = {}
        for device ∈ system do
            if device ∈ localDevices and device ∉ pinnedDevices then
                localSystem+ = device
            else
                bins[closestColorBin(device.color)]+ = device.intensity
        intensity.append(∑ localSystem.intensity)
        localSystems.append(localSystem)
    weights = normalizeWeights(colors.weights)
    results = ClosureOverUnevenBuckets(intensitiy, weights, colors, bins)

    /* Assign Colors to System */
    for i = 0 → localSystems.size do
        local = localSystems[i]
        for device ∈ local do
            deivce.color = results[i] + N(0, 0.1)
```

Algorithm 7: Closure Over Uneven Buckets

```
Function ClosureOverUnevenBuckets(object, bucket, colors, preFill)
    sumValues = 0
    closureDripOrder = []
    for i = 0 → object.size do
        sumValues+ = object[i]
        closureDripOrder[i] = i
    sumValues+ = ∑ preFill
    remainingBucketCapacity = [0] * bucket.size

    /* Remove prefilled bucket values from remaining capacity */
    remainingBucketCapacity = sumValues · bucket − preFill
    shuffle(closureDripOrder)
    currentBucket = 0

    /* Now drip into buckets */
    for i = 0 → object.size do
        if object[closureDripOrder[i]] < remainingBucketCapacity[currentBucket] then
            colors[closureDripOrder[i]] = currentBucket
            currentBucket = (currentBucket + 1)  mod bucket.size
        else
            colors[closureDripOrder[i]] = maxRemainingCapacityBucket(remainingBucketCapacity)
        remainingBucketCapacity[colors[closureDripOrder[i]]]− = object[closureDripOrder[i]]

    /* It's not entirely clear from this code right now but colors constains the id of the color
       that ends up getting assigned to the system at index i in the colors vector */
    return colors
```