

# Трасиране на лъчи 101

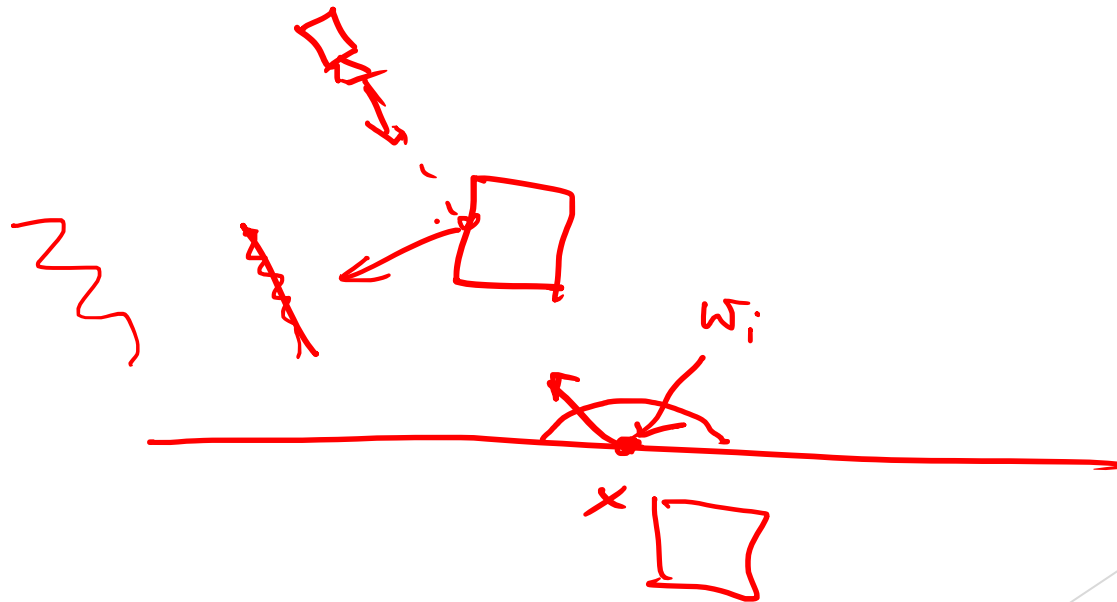
Андрей Дренски, 13.окт.2020г.

# Agenda

- ▶ защо трасираме лъчи и какви са те
- ▶ пресичане на лъч с 2D/3D примитива
- ▶ (ускорено) пресичане на лъч с прост 2D/3D обект
- ▶ (ускорено) пресичане на лъч с множество обекти
- ▶ получаване на допълнителна информация от пресичане на обект
  - ▶ нормали, загладени нормали, текстуриране
- ▶ трансформации, пресичане на лъч с трансформиран(и) обект(и)
- ▶ motion-blurred обекти, пресичане на лъч с MB обект(и)

и нека бъде

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i$$

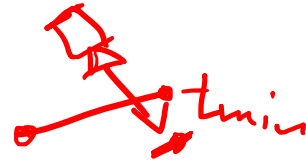


# Фундаментални типове и операции

```
struct Point {  
    float x, y;  
};  
struct Vector {  
    float x, y;  
};  
  
Point operator+(const Point& p, const Vector& v) { return { p.x + v.x, p.y + v.y }; }  
Vector operator-(const Point& p0, const Point& p1) { return { p0.x - p1.x, p0.y - p1.y }; }  
Vector operator*(const Vector& v, const float k) { return { v.x*k, v.y*k }; }  
Vector normalize(const Vector& v) {  
    const float d = std::sqrt(v.x*v.x + v.y*v.y);  
    return { v.x / d, v.y / d };  
}  
  
struct Segment {  
    Point p0, p1;  
};
```

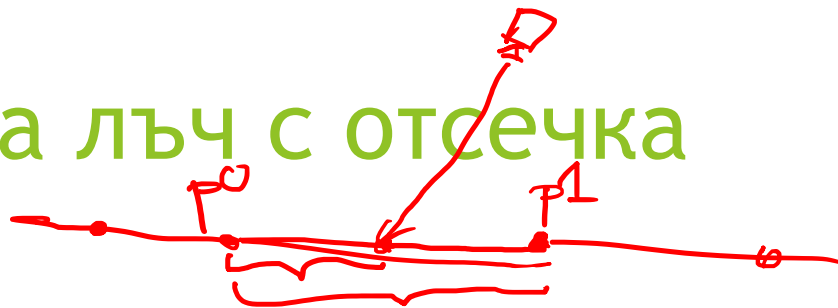
# Фундаментални типове и операции

```
struct Ray {  
    Point origin;  
    Vector dir;  
    Ray(const Point& origin, const Vector& dir)  
        : origin{ origin }, dir{ normalize(dir) } {}  
};  
  
struct Segment {  
    Point p0, p1;  
    SegmentHit intersect(const Ray& ray,  
        const float tmin = 0.f, const float tmax = 1e18f) const;  
};  
  
const float inf = std::numeric_limits<float>::infinity();  
  
bool approx(const float& x, const float& y) { return (std::abs(x - y) < 1e-6f); }
```



# Пресичане на лъч с отсечка

```
struct SegmentHit {  
    float t, u;  
  
    SegmentHit() : t{ inf } {}  
    bool isValid() const { return (t < inf); }  
    operator bool() const { return isValid(); }  
};
```

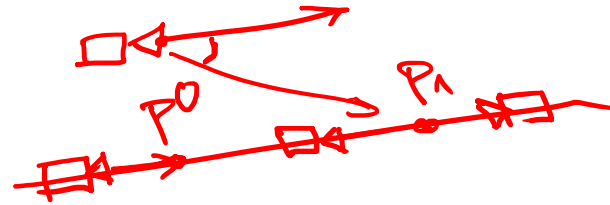


# Пресичане на лъч с отсечка

```
struct SegmentHit {
    float t, u;

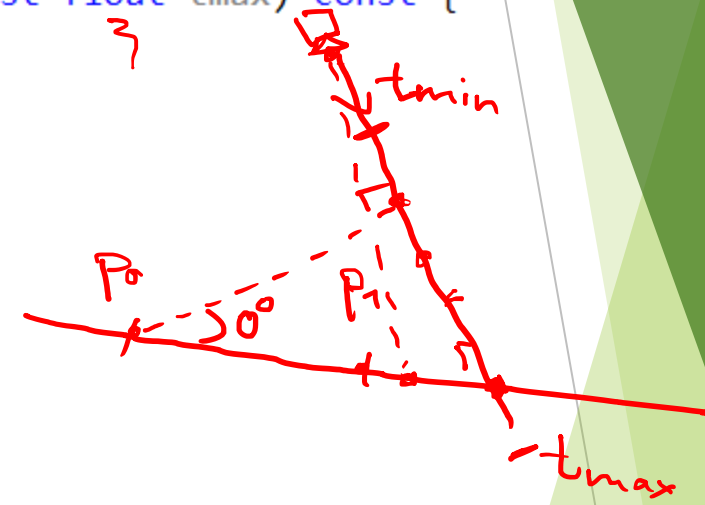
    SegmentHit() : t{ inf } {}
    bool isValid() const { return (t < inf); }
    operator bool() const { return isValid(); }
};

SegmentHit Segment::intersect(const Ray& ray, const float tmin, const float tmax) const {
    const Vector w = p0 - ray.origin;
    const Vector segDir = p1 - p0;
    // same sign as cross-product, i.e. 0 if parallel
    const float denom = (ray.dir.x*segDir.y - ray.dir.y*segDir.x);
    if (approx(denom, 0.f)) {
        const bool collinear = approx(w.x*segDir.y - w.y*segDir.x, 0.f);
        if (!collinear) {
            return {};
        }
        // some stupid edge cases here
        vassert(false && "to-do");
        return {};
    }
}
```



# Пресичане на лъч с отсечка

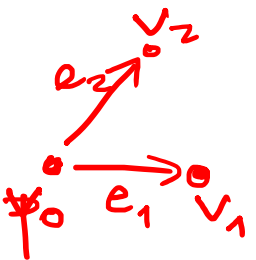
```
SegmentHit Segment::intersect(const Ray& ray, const float tmin, const float tmax) const {  
    const Vector w = p0 - ray.origin;  
    const Vector segDir = p1 - p0;  
    // same sign as cross-product, i.e. 0 if parallel  
    const float denom = (ray.dir.x*segDir.y - ray.dir.y*segDir.x);  
    if (approx(denom, 0.f)) { ... }  
    const float u = (ray.dir.y*w.x - ray.dir.x*w.y) / denom;  
    if (u < 0.f || u > 1.f) {  
        return {};  
    }  
    const float t = -(segDir.x*w.y - segDir.y*w.x) / denom;  
    return ((t >= tmin && t < tmax) ? SegmentHit{ t,u } : SegmentHit{});  
}
```





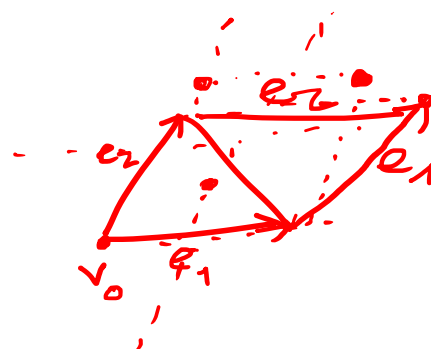
# Пресичане на лъч с триъгълник (3D)

```
struct TriangleHit {  
    float t, u, v;  
    z  
    TriangleHit() : t{ inf } {}  
    bool isValid() const { return (t < inf); }  
    operator bool() const { return (t < inf); }  
};  
  
struct Triangle {  
    Point v0;  
    Vector e1, e2;  
  
    TriangleHit intersect(const Ray& ray,  
        const float tmin = 0.f, const float tmax = 1e18f) const;  
};
```

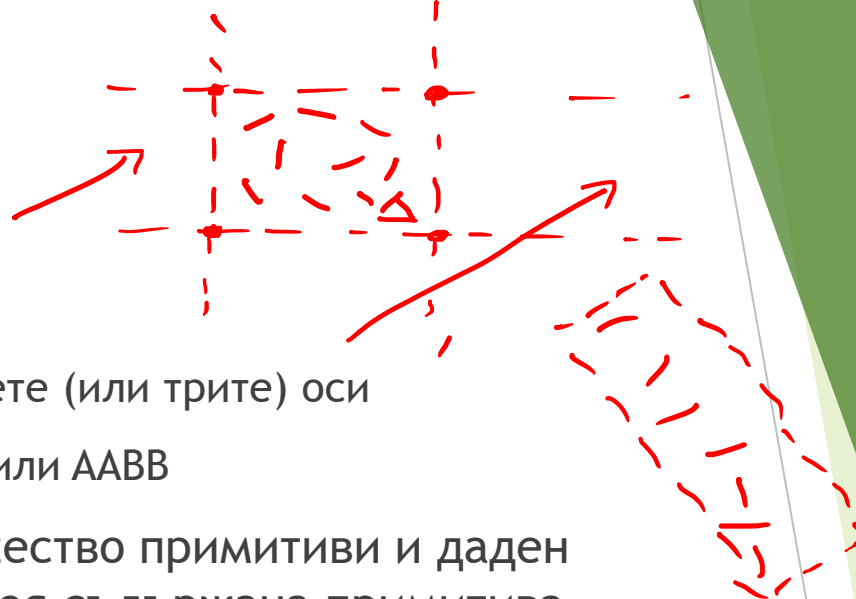


# Пресичане на лъч с триъгълник (3D)

```
TriangleHit Triangle::intersect(const Ray& ray, const float tmin, const float tmax) const {  
    // First intersect the ray with the triangle's plane  
    const Vector n = normalize(cross(e1, e2));  
    const float denom = dot(n, ray.dir);  
    // Ray & plane are collinear or parallel  
    if (approx(denom, 0.f)) { ... }  
    const float t = dot(n, v0 - ray.origin) / denom;  
    if (t < tmin || t >= tmax) {  
        return {};  
    }  
    // The intersection point  
    const Point inters = ray.origin + t*ray.dir;  
    const float e1e2 = dot(e1, e2);  
    const float e1e1 = dot(e1, e1);  
    const float e2e2 = dot(e2, e2);  
    const float denom2 = e1e2*e1e2 - e1e1*e2e2;  
    vassert(!approx(denom2, 0.f)); // Triangle is degenerate  
    const Vector w = inters - v0;  
    const float u = (e1e2*dot(w, e2) - e2e2*dot(w, e1)) / denom2;  
    const float v = (e1e2*dot(w, e1) - e1e1*dot(w, e2)) / denom2;  
    return ((u >= 0 && v >= 0 && u + v <= 1) ? TriangleHit{ t, u, v } : TriangleHit{});  
}
```



# much ado about boxes



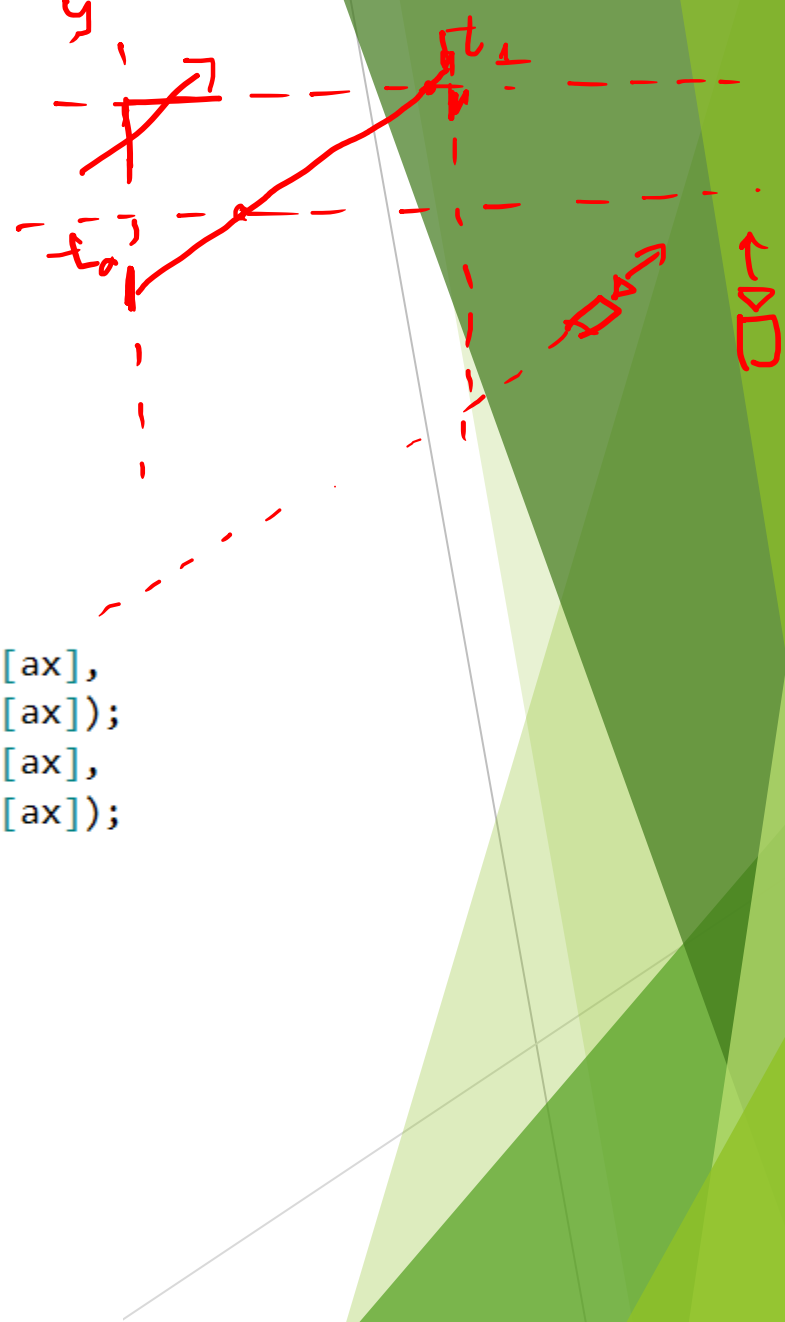
- ▶ друга важна примитива е bounding box-ът
  - ▶ „кутия“, зададена от числови интервали по двете (или трите) оси
  - ▶ оттам известна като axis-aligned bounding box, или AABB
- ▶ идея: ако кутия съдържа изцяло дадено множество примитиви и даден лъч не я пресече, то той няма да пресече никоя съдържаща примитива

```
class Box {  
    Point pmin, pmax;  
public:  
    Box() : pmin{ 1e18f, 1e18f }, pmax{ -1e18f, -1e18f } {}  
    void expand(const Point& p) {  
        pmin = { std::min(pmin.x, p.x), std::min(pmin.y, p.y) };  
        pmax = { std::max(pmax.x, p.x), std::max(pmax.y, p.y) };  
    }  
    void expand(const float k) {  
        const Point mid = midpoint(pmin, pmax);  
        pmin += (mid - pmin)*k;  
        pmax += (pmax - pmin)*k;  
    }  
    float intersect(const Ray& ray, float tmin = 0.f, float tmax = 1e18f) const;  
};
```

# much ado about boxes

```
enum class Axis { X, Y };
```

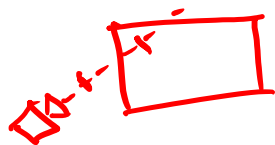
```
float Box::intersect(const Ray& ray, float tmin, float tmax) const {  
    // Whether the first wall (relative to ray direction) is hit  
    bool hitN = false;  
    for (const Axis ax : {Axis::X, Axis::Y}) {  
        // Note: safe to do divisions by zero;  
        // the resulting inf-s will get filtered out by the min/max  
        const float t0 = std::min((pmin[ax] - ray.origin[ax]) / ray.dir[ax],  
                                   (pmax[ax] - ray.origin[ax]) / ray.dir[ax]);  
        const float t1 = std::max((pmin[ax] - ray.origin[ax]) / ray.dir[ax],  
                                   (pmax[ax] - ray.origin[ax]) / ray.dir[ax]);  
  
        //tmin = std::max(t0, tmin);  
        if (t0 > tmin) {  
            tmin = t0;  
            hitN = true;  
        }  
        tmax = std::min(t1, tmax);  
        if (tmax <= tmin) {  
            return inf;  
        }  
    }  
    return (hitN ? tmin : tmax);  
}
```



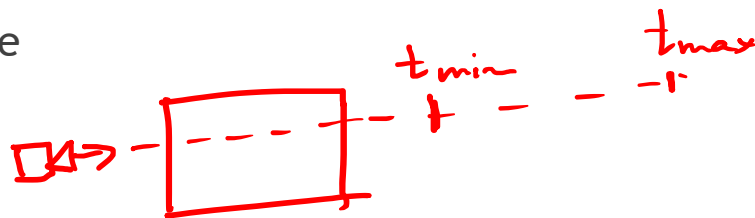
# much ado about boxes

- ▶ примери за взаимно разположение на „ограничен“ лъч с кутия:

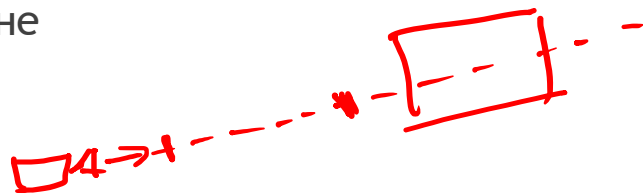
- ▶ класическо пресичане



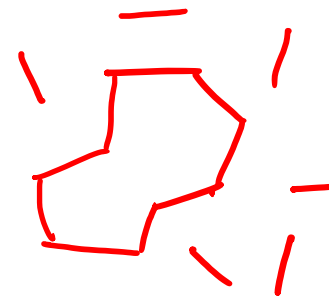
- ▶ надхвърляне



- ▶ недостигане



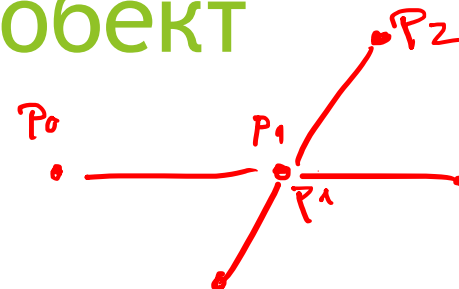
# Какво представлява един обект



- ▶ по най-простата (и обща) дефиниция - множество от примитиви
  - ▶ важно: последователността на върховете на примитивата има значение!
  - ▶ тя задава ориентацията на примитивата
- ▶ въпрос на представяне, например:

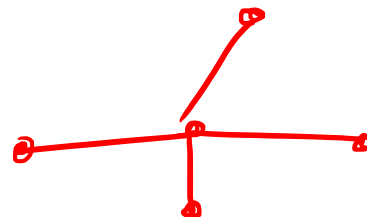
```
class Mesh {  
    std::vector<Segment> segments;  
public:  
    int numPrimitives() const { return segments.size(); };  
    const Segment& getPrimitive(const int idx) const { return segments[idx]; }  
};
```

# Какво представлява един обект



- ▶ недостатък: пазим всеки връх по два пъти
  - ▶ защо не пазим масив от върховете в последователността, в която образуват многоъгълника?

# Какво представлява един обект



- ▶ недостатък: пазим всеки връх по два пъти
  - ▶ защо не пазим масив от върховете в последователността, в която образуват многоъгълника?
- ▶ има друго решение: индексни буфери
  - ▶ another level of indirection, но спестената памет си струва

```
class Mesh {  
    std::vector<Point> vertices; → 5  
    std::vector<int> indices; → 4 * 2  
    Box box; // Constructed during parsing  
public:  
    int numPrimitives() const { return indices.size() / 2; }  
    Segment getPrimitive(const int idx) const {  
        return { vertices[indices[2*idx]],  
                vertices[indices[2*idx + 1]] };  
    }  
};
```




# Какво представлява един обект

- ▶ недостатък: пазим всеки връх по два пъти
  - ▶ защо не пазим масив от върховете в последователността, в която образуват многоъгълника?
- ▶ има друго решение: индексни буфери
  - ▶ another level of indirection, но спестената памет си струва

```
class Mesh {  
    std::vector<Point> vertices;  
    std::vector<int> indices;  
    Box box; // Constructed during parsing  
public:  
    int numPrimitives() const { return indices.size() / 2; }  
    Segment getPrimitive(const int idx) const {  
        return { vertices[indices[2*idx]],  
                vertices[indices[2*idx + 1]] };  
    }  
};
```

# Пресичане на лъч с обект

```
struct Hit {  
    int primIdx;   
    float t;  
    float u;  
  
    Hit() : t{ inf } {}  
    bool isValid() const { return (t < inf); }  
    operator bool() const { return (t < inf); }
```

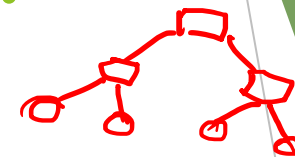
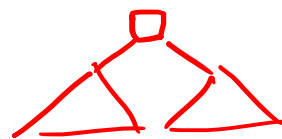
# Пресичане на лъч с обект

```
struct Hit {  
    int primIdx;  
    float t;  
    float u;  
  
    Hit() : t{ inf } {}  
    bool isValid() const { return (t < inf); }  
    operator bool() const { return (t < inf); }  
  
    void improveBy(const Hit& other) {  
        if (other && other.t < t) {  
            *this = other;  
        }  
    }  
  
    void improveBy(const int primIdx, const SegmentHit& other) {  
        if (other && other.t < t) {  
            this->primIdx = primIdx;  
            t = other.t;  
            u = other.u;  
        }  
    }  
};
```

# Пресичане на лъч с обект

```
Hit Mesh::intersect(const Ray& ray, const float tmin, const float tmax) const {  
    if (!box.intersect(ray, tmin, tmax)) {  
        return {};  
    }  
    Hit bestHit;  
    for (int i = 0; i < numPrimitives(); ++i) {  
        bestHit.improveBy(i, getPrimitive(i).intersect(ray, tmin, tmax));  
        // We can bump tmin and tmax on successful improvement,  
        // but there's no performance benefit of doing it.  
    }  
    return bestHit;  
}
```

# Ускорено пресичане на лъч с обект: kd-tree



- ▶ като двоично дърво за търсене, но в 2D/3D
- ▶ част от групата на space-partitioning trees, като quadtree/octree/BVH tree
  - ▶ характерна за всички е разликата между вътрешни възли и листа
- ▶ всеки вътрешен възел разделя пространството на две допълващи се подпространства, и има два възела-наследници
- ▶ всяко листо съдържа множество примитиви (не е задължително да е един)

```
struct Node {  
    const bool isLeaf;  
    float splitVal;  
    int left, right;  
  
    Node(const int left, const int right)  
        : isLeaf{ true }, left{ left }, right{ right } {}  
    Node(const float splitVal)  
        : isLeaf{ false }, splitVal{ splitVal } {}  
};
```

# Ускорено пресичане на лъч с обект: kd-tree



```
class Kdtree {  
    const Mesh geometry;  
    // All tree nodes. Root node is [0], non-leaf nodes contain child node indices in this array.  
    // The geometry's bounding box is used to obtain each node's implicit bounding box during traversal.  
    std::vector<Node> nodes;  
    // Contains reordered primitive indices. Each leaf node points to a range in this array.  
    // As an alternative, we can rearrange the geometry's internal index buffer.  
    std::vector<int> primIndices;  
    // Construction parameters  
    static const int maxDepth = 60;  
    static const int maxPrimsInLeaf = 15;  
};
```

5, 3, 0

# Ускорено пресичане на лъч с обект: kd-tree

- рекурсивно построяване на kd-дърво по дадено множество от примитиви

```
int buildRecursive(const int from, const int to, const int depth);
public:
    Kdtree(Mesh&& geometry_) : geometry{ std::move(geometry_) } {
        const int numPrimitives = geometry.numPrimitives();
        primIndices.resize(numPrimitives);
        // Build our index array (not to be confused with the mesh's one)
        for (int i = 0; i < numPrimitives; ++i) {
            primIndices[i] = i;
        }
        buildRecursive(0, numPrimitives, 0); // This returns 0, i.e. the root node index
    }
};
```

# Ускорено пресичане на лъч с обект: kd-tree

- рекурсивно построяване на kd-дърво по дадено множество от примитиви

```
// Invariant: constructs a node, managing the primitives, indexed by the range primIndices[from;to).  
// May rearrange the values in this range. Adds a single node to the array & returns its index.  
int Kdtree::buildRecursive(const int from, const int to, const int depth) {  
    if (depth >= maxDepth || (to - from) <= maxPrimsInLeaf) {  
        nodes.push_back({ from, to });  
        return (nodes.size() - 1); // The index of the freshly inserted node  
    }  
    const Axis ax = (depth % 2 ? Axis::Y : Axis::X);  
    const float splitVal = getMidPoint(from, to, ax);  
    const int mid = partition(from, to, ax, splitVal);  
    nodes.push_back({ splitVal }); // Insert before the recursive calls (!)  
    const int nodeId = nodes.size() - 1; // The index of the freshly inserted node  
    // Do not make a local Node& node = nodes[nodeId] - it'll become dangling after vector reallocations!  
    nodes[nodeId].left = buildRecursive(from, mid, depth + 1); // Note: always returns nodeId+1  
    nodes[nodeId].right = buildRecursive(mid, to, depth + 1);  
    return nodeId;  
};
```



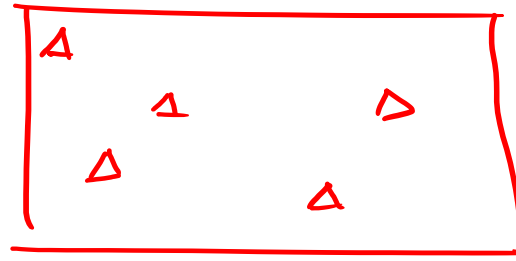
# Ускорено пресичане на лъч с обект: kd-tree



Липсващи части:

- ▶ `getMidPoint()` - намиране на подходяща точка (задаваща ос), разбиваща множеството на достатъчно близки по големина части
  - ▶ има апроксимиращи алгоритми, чиято специалност е да са *good enough*<sup>™</sup>, напр. Median-of-medians (за повече → доц. Минко Марков след две седмици)
- ▶ `partition()` - разбиване на множеството спрямо намерената ос
  - ▶ проблем: винаги може някоя примитива да е и от двете страни на оста
  - ▶ ако са отсечки, можем да ги разцепим на две - но тогава броят им се увеличава
  - ▶ но триъгълник в общия случай не се разцепва на два триъгълника
  - ▶ може да включим примитивата и в двата наследника
    - ▶ това поражда още проблеми...
  - ▶ винаги може дадена примитива да лежи на избраната ос...

# Ускорено пресичане на лъч с обект: kd-tree



- аналогично рекурсивно пресичане на лъч с kd-дърво

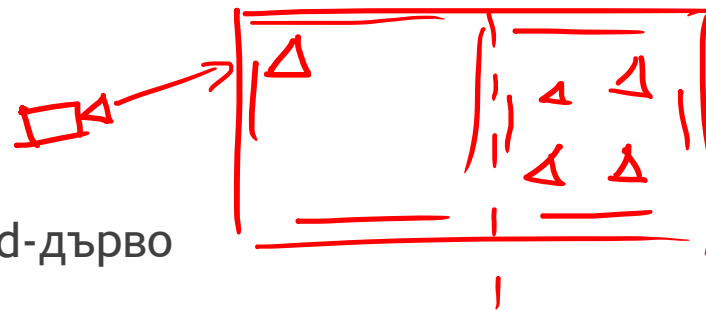
```
Hit Kdtree::intersect(const Ray& ray, const float tmin, const float tmax) const {  
    // Modifying [tmin;tmax] here might not be needed here  
    if (geometry.box.intersect(ray, tmin, tmax)) {  
        return intersectRecursive(ray, geometry.box, tmin, tmax, 0, 0);  
    } else {  
        return {};  
    }  
}
```

# Ускорено пресичане на лъч с обект: kd-tree

- аналогично рекурсивно пресичане на лъч с kd-дърво

```
// Invariant: box is the AABB for all primitives in nodes[nodeIdx],
// and we have already tested whether ray intersects with it.
Hit Kdtree::intersectRecursive(
    const Ray ray, const Box box, const float tmin,
    const float tmax, const int nodeId, const int depth
) const {
    const Node& node = nodes[nodeId];
    if (node.isLeaf) {
        Hit bestHit;
        // there would be less confusion if this was a range or std::span :)
        for (int i = node.left; i < node.right; ++i) {
            const int primIdx = primIndices[i];
            bestHit.improveBy(primIdx, geometry.getPrimitive(primIdx).intersect(ray, tmin, tmax));
        }
        return bestHit;
    } else {
```

# Ускорено пресичане на лъч с обект: kd-tree



- аналогично рекурсивно пресичане на лъч с kd-дърво

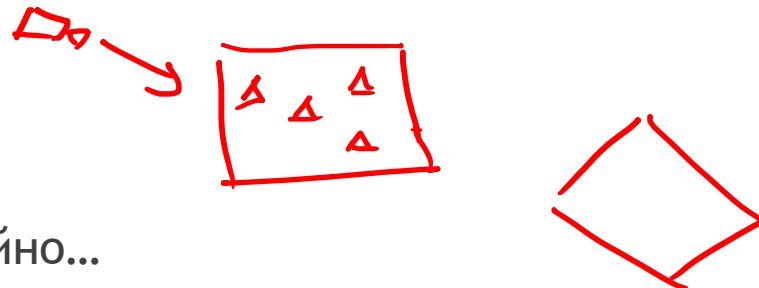
```
} else {  
    const Axis ax = (depth % 2 ? Axis::Y : Axis::X);  
    const auto [leftBox, rightBox] = box.split(ax, node.splitVal);  
    // Try intersecting with the nearest box (and on success, its primitives) first.  
    // If there is a hit with some primitive inside it, there's no point in intersecting the other box.  
    // Important: this is guaranteed by the fact that the two bounding boxes do not overlap!  
    const bool leftIsNearest = (ray.dir[ax] > 0);  
    const int nearIdx = (leftIsNearest ? node.left : node.right);  
    const int farIdx = (leftIsNearest ? node.right : node.left);  
    const Box& nearBox = (leftIsNearest ? leftBox : rightBox);  
    const Box& farBox = (leftIsNearest ? rightBox : leftBox);  
}
```

# Ускорено пресичане на лъч с обект: kd-tree

- аналогично рекурсивно пресичане на лъч с kd-дърво

```
// Also, if the ray is entirely before/beyond the split
// line, we can skip intersecting one of the boxes.
Hit bestHit;
float boxHit = nearBox.intersect(ray, tmin, tmax);
if (boxHit < inf) {
    // Note: it is beneficial to use the box intersection result here
    // to limit the [tmin, tmax] range here. In order to do this, we
    // need to know whether a given box is intersected once or twice (!)
    bestHit.improveBy(intersectRecursive(ray, nearBox, tmin, tmax, nearIdx, depth + 1));
} else {
    boxHit = farBox.intersect(ray, tmin, tmax);
    if (boxHit < inf) {
        // Same comment about modifying [tmin;tmax] here, too
        bestHit.improveBy(intersectRecursive(ray, farBox, tmin, tmax, farIdx, depth + 1));
    }
}
return bestHit;
```

# Пресичане на лъч с множество обекти



- ▶ винаги можем да обходим всеки обект линейно...
- ▶ алтернатива: обединяване на всички примитиви на всички обекти в едно kd-дърво
  - ▶ всяка примитива трябва да си знае ~~къде~~ ~~къде~~ обекта, от който е дошла

# Ускорено пресичане на лъч с множество обекти: BVH-tree

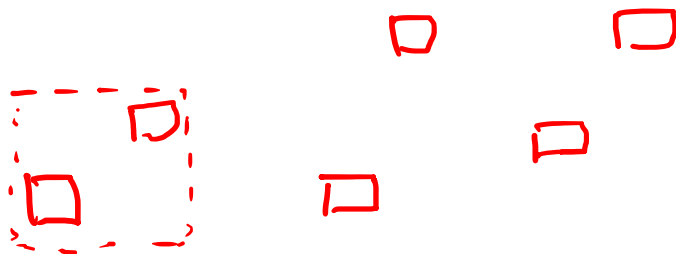
- ▶ BVH = Bounding Volume Hierarchy
- ▶ едно от по-сложните space partitioning дървета
- ▶ всеки обект в сцената е „примитива“ за това дърво, представена от своята кутия (bounding box)
- ▶ всеки вътрешен възел в дървото представлява кутия, получен от обединението на две други кутии (на каквито и да е възли)



```
Box(const Box& b1, const Box& b2) {  
    pmin.x = std::min(b1.pmin.x, b2.pmin.x);  
    pmin.y = std::min(b1.pmin.y, b2.pmin.y);  
    pmax.x = std::max(b1.pmax.x, b2.pmax.x);  
    pmax.y = std::min(b1.pmax.y, b2.pmax.y);  
}
```

# Ускорено пресичане на лъч с множество обекти: BVH-tree

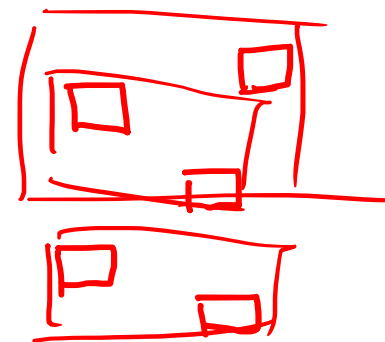
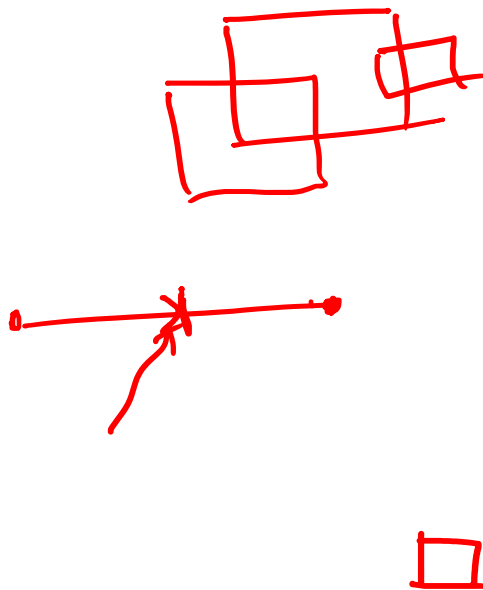
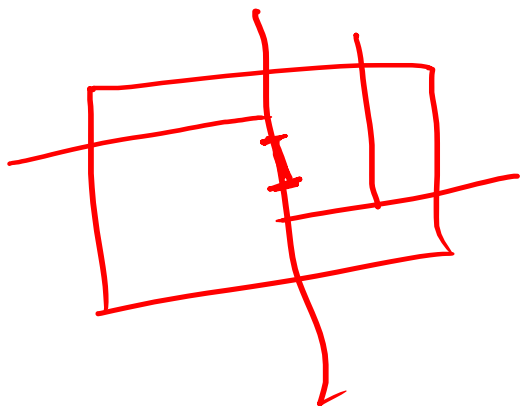
- ▶ алтернативи за построение на BVH дървета:
  - ▶ top-down е аналогичен на kd-дървото; нетривиален и генерира лоши дървета
  - ▶ подлежи на паралелизиране
- ▶ bottom-up: групиране/клъстеризиране на близки обекти във възможно най-малки обединяващи ги bounding box-ове
- ▶ често с по-висока сложност, паралелизирането не намалява сложността





# Ускорено пресичане на лъч с множество обекти: BVH-tree

- получаваме „безплатен“ dimensionality curse - примитивите могат да разположени взаимно по много повече начини



# Окончателно представяне на света

- ▶ двуслойно дърво-от-дървета: BVH, в който всяко листо съдържа kd-дърво
- ▶ компромис между бързи промени и ефикасно пресичане
  - ▶ значително предимство при инстанцирани обекти (more on that later)
- ▶ пресичането и на двете дървета е логаритмично по броя примитиви

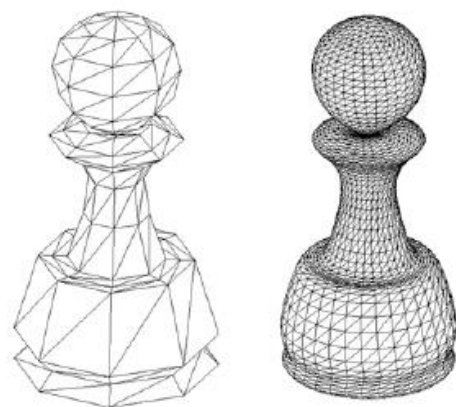
# Получаване на допълнителна информация при пресичане

- ▶ след установяване на най-близката пресечна точка на лъча, имаме нужда от повече информация за качествено оцветяване shade-ване
- ▶ геометрична нормала, т.е. ориентацията на примитивата:

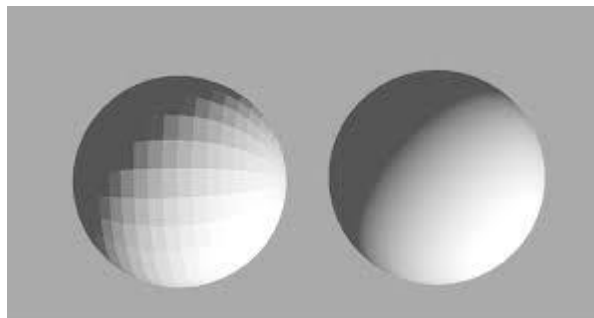
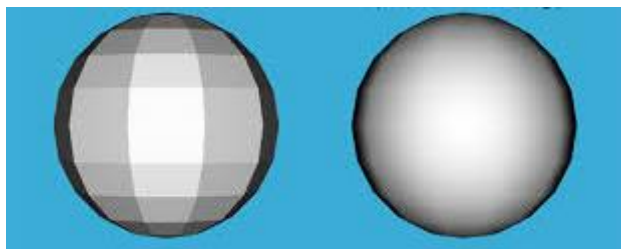
```
[-] Vector Segment::geomNormal() const {  
    |     return normalize({ p0.y - p1.y, p1.x - p0.x });  
    | }  
  
[-] Vector Triangle::geomNormal() const {  
    |     return normalize(cross(e1, e2));  
    | }
```

# Получаване на допълнителна информация при пресичане

- ▶ понякога искаме обектите да изглеждат загладени
  - ▶ твърде фино разбиване на геометрията е скъпо откъм време и памет



- ▶ можем да загладим изкуствено обектите



# Получаване на допълнителна информация при пресичане

- ▶ всеки връх носи собствена нормала
  - ▶ при пресичане на дадена отсечка, интерполираме (внимателно) между нормалите в краищата ѝ с бари-координатата на пресечната точка

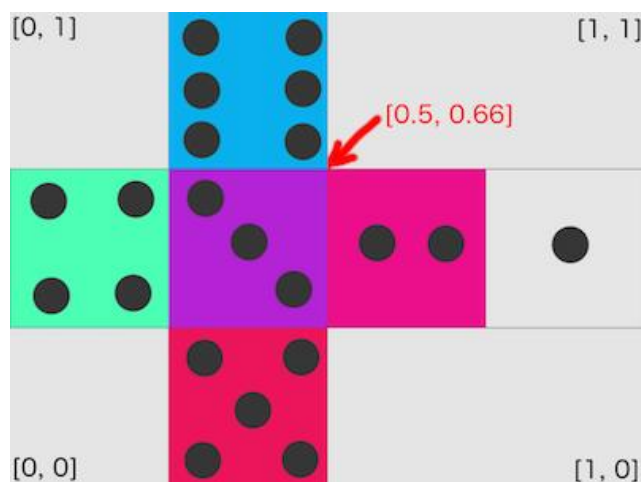
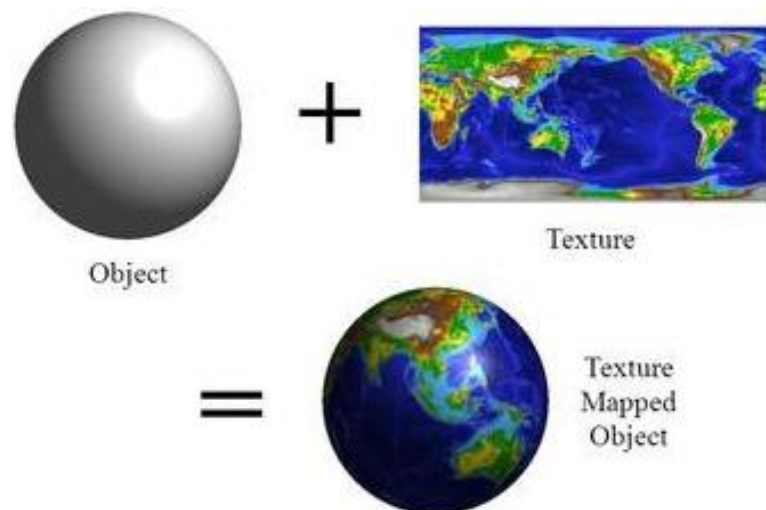
```
Vector Mesh::getNormal(const int idx, const float u) const {  
    return normalize(vertexNormals[indices[2*idx  ]]*u  
        + vertexNormals[indices[2*idx+1]]*(1-u));  
}
```

- ▶ всеки връх на всеки триъгълник има собствена нормала
  - ▶ индексни буфери to the rescue

```
Vector Mesh::getNormal(const int idx, const float u) const {  
    return normalize(normals[normalIndices[2*idx  ]]*u  
        + normals[normalIndices[2*idx+1]]*(1-u));  
}
```

# Получаване на допълнителна информация при пресичане

- ▶ най-важното - текстуриране
  - ▶ тривиално за 2D обекти, затова демонстрираме за 3D
- ▶ отново за всеки връх на всеки триъгълник имаме координати в UV-пространството на текстурата
- ▶ получаваме „наslagване“ на част от текстурата върху дадена примитива



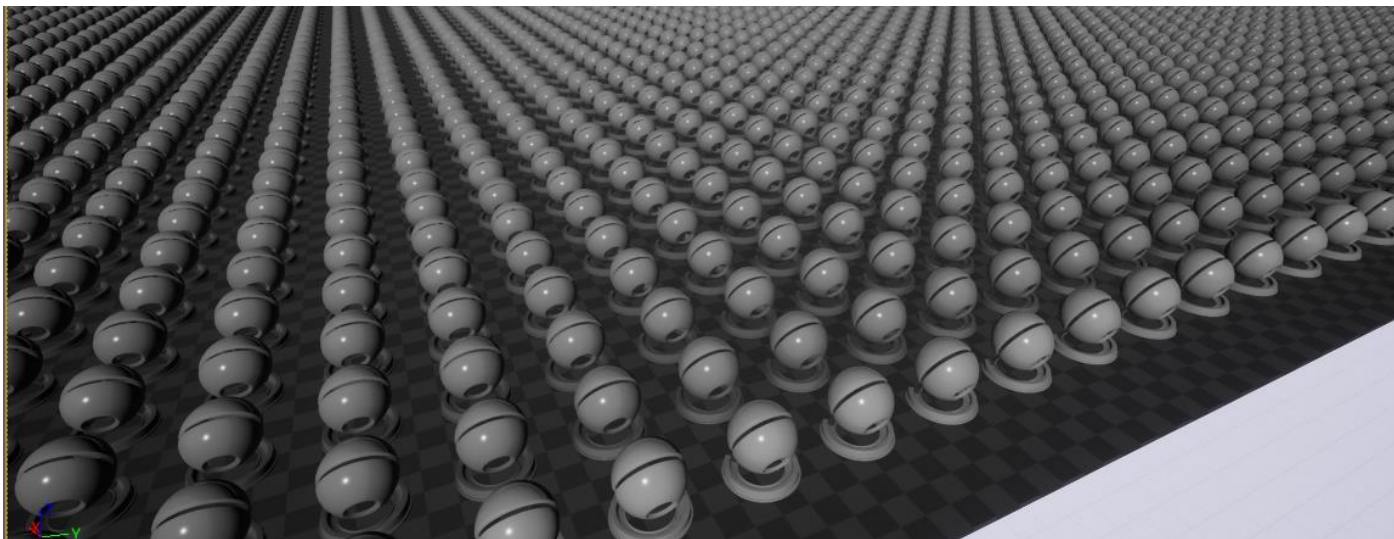
# Получаване на допълнителна информация при пресичане

- ▶ най-важното - текстуриране
  - ▶ тривиално за 2D обекти, затова демонстрираме за 3D
- ▶ отново за всеки връх на всеки триъгълник имаме координати в UV-пространството на текстурата
- ▶ при пресичане с даден триъгълник отново интерполираме

```
struct UVCoord {  
    float u, v;  
};  
UVCoord Mesh::getUVs(const int idx, const float u, const float v) const {  
    return uvs[uvIndices[3*idx  ]]*(1-u-v)  
        + uvs[uvIndices[3*idx+1]]*u  
        + uvs[uvIndices[3*idx+2]]*v;  
}
```

# Трансформирание и инстанциране на обекти

- ▶ може да искаме да дублираме даден обект, без да променяме копието
  - ▶ напр. само да го преместим
  - ▶ това се нарича инстанциране - създаваме reference, вместо пълно копие





# Трансформиране и инстанциране на обекти

- ▶ може да искаме да дублираме даден обект, без да променяме копието
  - ▶ напр. само да го преместим
  - ▶ това се нарича инстанциране - създаваме reference, вместо пълно копие
- ▶ щом говорим за „преместване“, трябва всеки обект да има идея за „положение“ → това е вектор на трансляция
- ▶ взаимното разположение на върховете не се променя → няма нужда да ги променяме
- ▶ идея: обектите и примитивите им живеят в различни пространства

```
enum class Space { World, Object };  
  
template <Space S>  
struct Point {  
    float x, y;  
};
```

# Трансформиране и инстанциране на обекти

- идея: обектите и примитивите им живеят в различни пространства

```
template <Space S>
struct Vector {
    float x, y;
};

template <Space S>
Point<S> operator+(const Point<S>& p, const Vector<S>& v)
{ return { p.x + v.x, p.y + v.y }; }

template <Space S>
struct Ray {
    Point<S> origin;
    Vector<S> dir;
    Ray(const Point<S>& origin, const Vector<S>& dir)
        : origin{ origin }, dir{ normalize(dir) } {}
};
```

# Трансформирание и инстанциране на обекти

- идея: обектите и примитивите им живеят в различни пространства

```
template <Space S1, Space S2>
struct Translation {
    float x, y;

    Point<S2> apply(const Point<S1>& v) { return { v.x + x, v.y + y }; }
    Vector<S2> apply(const Vector<S1>& v) { return { v.x, v.y }; }

    Point<S1> applyInverse(const Point<S2>& v) { return { v.x - x, v.y - y }; }
    Vector<S1> applyInverse(const Vector<S2>& v) { return { v.x, v.y }; }
};
```

# Трансформиране и инстанциране на обекти

- идея: обектите и примитивите им живеят в различни пространства

```
struct Mesh {  
    std::vector<Point<Space::Object>> vertices;  
    std::vector<Vector<Space::Object>> vertexNormals; //  
    std::vector<int> indices;  
  
    Box<Space::World> box; // Constructed during parsing  
    Translation<Space::Object, Space::World> transl;  
public:  
    Hit intersect(const Ray<Space::World>& ray,  
        const float tmin = 0.f, const float tmax = 1e18f) const;
```

# Трансформиране и инстанциране на обекти

- идея: обектите и примитивите им живеят в различни пространства

```
Hit Mesh::intersect(const Ray<Space::World>& ray, const float tmin, const float tmax) const {  
    if (!box.intersect(ray, tmin, tmax)) {  
        return {};  
    }  
    const Ray<Space::Object> rayObj{ transl.applyInverse(ray.origin), transl.applyInverse(ray.dir) };  
    Hit bestHit;  
    for (int i = 0; i < numPrimitives(); ++i) {  
        bestHit.improveBy(i, getPrimitive(i).intersect(rayObj, tmin, tmax));  
        // We can bump tmin and tmax on successful improvement,  
        // but there's no performance benefit of doing it.  
    }  
    return bestHit;  
}
```

# Трансформиране и инстанциране на обекти

- ▶ същата концепция можем да приложим за други видове трансформации:
  - ▶ ротация
  - ▶ скалиране
  - ▶ shearing
  - ▶ комбинация от гореописаните

```
template<Space S1, Space S2>  
struct Matrix {  
    float m[2][2];  
};
```

# Трансформиране и инстанциране на обекти

- ▶ същата концепция можем да приложим за други видове трансформации:
  - ▶ ротация
  - ▶ скалиране
  - ▶ shearing
  - ▶ комбинация от гореописаните

```
template <Space S1, Space S2>
Point<S2> operator*(const Matrix<S1, S2>& mat, const Point<S1>& p) {
    return { { mat.m[0][0]*p.x + mat.m[0][1]*p.y },
             { mat.m[1][0]*p.x + mat.m[1][1]*p.y } };
}
template <Space S1, Space S2>
Vector<S2> operator*(const Matrix<S1, S2>& mat, const Vector<S1>& p) {
    return { { mat.m[0][0]*p.x + mat.m[0][1]*p.y },
             { mat.m[1][0]*p.x + mat.m[1][1]*p.y } };
}
```

# Трансформиране и инстанциране на обекти

- ▶ трансляцията е отделна и трябва да внимаваме в каква последователност я комбинираме с другите трансформации (!)
- ▶ окончателно имаме:

```
template <Space S1, Space S2>
struct Transform {
    static_assert(S1 != S2);
    float m[2][2]; // This can be Matrix<S1, S2>
    float off[2];  // This can be Vector<S2>. ...
};
```



# Трансформиране и инстанциране на обекти

- ▶ трансляцията е отделна и трябва да внимаваме в каква последователност я комбинираме с другите трансформации (!)

```
// Post-multiplication of matrix & column-vector
template <Space S1, Space S2>
Point<S2> operator*(const Transform<S1, S2>& tm, const Point<S1>& p) {
    // This can be tm.m*p + tm.off;
    return { tm.m[0][0]*p.x + tm.m[0][1]*p.y + tm.off[0],
            tm.m[1][0]*p.x + tm.m[1][1]*p.y + tm.off[1] };
}

template <Space S1, Space S2>
Vector<S2> operator*(const Transform<S1, S2>& tm, const Vector<S1>& p) {
    // This can be just tm.m*p
    return { tm.m[0][0]*p.x + tm.m[0][1]*p.y,
            tm.m[1][0]*p.x + tm.m[1][1]*p.y };
}
```

# Трансформиране и инстанциране на обекти

- ▶ трансляцията е отделна и трябва да внимаваме в каква последователност я комбинираме с другите трансформации (!)

```
struct Mesh {  
    std::vector<Point<Space::Object>> vertices;  
    std::vector<Vector<Space::Object>> vertexNormals; //  
    std::vector<int> indices;  
  
    Box<Space::World> box; // Constructed during parsing  
    Transform<Space::Object, Space::World> tm;  
    Transform<Space::World, Space::Object> itm;  
public:  
    Hit intersect(const Ray<Space::World>& ray,  
        const float tmin = 0.f, const float tmax = 1e18f) const;
```

# Трансформиране и инстанциране на обекти

- ▶ трансляцията е отделна и трябва да внимаваме в каква последователност я комбинираме с другите трансформации (!)

```
Hit Mesh::intersect(const Ray<Space::World>& ray, const float tmin, const float tmax) const {
    if (!box.intersect(ray, tmin, tmax)) {
        return {};
    }
    const Ray<Space::Object> rayObj{ itm*ray.origin, itm*ray.dir };
    Hit bestHit;
    for (int i = 0; i < numPrimitives(); ++i) {
        bestHit.improveBy(i, getPrimitive(i).intersect(rayObj, tmin, tmax));
        // We can bump tmin and tmax on successful improvement,
        // but there's no performance benefit of doing it.
    }
    // Now apply tm to whatever results we extract in object-space (f.e. normals)
    return bestHit;
}
```

# Трансформиране и инстанциране на обекти

- ▶ трансляцията е отделна и трябва да внимаваме в каква последователност я комбинираме с другите трансформации (!)
- ▶ как получаваме обратната трансформация? Защо?

```
template <Space S1, Space S2>
Transform<S2, S1> inverse(const Transform<S1, S2>& tm) {
    Transform<S2, S1> res;
    const float d = tm.m[0][0]*tm.m[1][1] - tm.m[0][1]*tm.m[1][0];
    res.m = { { tm.m[1][1]/d, -tm.m[0][1]/d},
              {-tm.m[1][0]/d, tm.m[0][0]/d} };
    // Note: this is the same as multiplying the negative
    // (i.e. inverse) offset by the inverse matrix
    res.off[0] = -(tm.off[0]*res.m[0][0] + tm.off[1]*res.m[0][1]);
    res.off[1] = -(tm.off[0]*res.m[1][0] + tm.off[1]*res.m[1][1]);
    return res;
}
```

Motion-blurred обекти:  
a jump into time-space

# Further reading

- ▶ Raytracing in one weekend / the next week / the rest of your life: <https://github.com/RayTracing/raytracing.github.io>
- ▶ Physically-based raytracing: <https://pbrt.org/>
- ▶ NVIDIA OptiX: <https://developer.nvidia.com/optix>
- ▶ Code samples: <https://github.com/Andreshk/MNKRaytracing>