INTERNATIONAL STANDARD

ISO/IEC 14496-19

First edition
2004-07-01

Information technology — Coding of audio-visual objects —

Part 19: **Synthesized texture stream**

Technologies de l'information — Codage des objets audiovisuels —

Partie 19: Flux de texture synthétisé

Reference number
ISO/IEC 14496-19:2004(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 14496-19 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 14496 consists of the following parts, under the general title *Information technology — Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

— *Part 7: Optimized reference software for coding of audio-visual objects*

— *Part 8: Carriage of ISO/IEC 14496 contents over IP networks*

— *Part 9: Reference hardware description*

— *Part 10: Advanced Video Coding*

— *Part 11: Scene description and application engine*

— *Part 12: ISO base media file format*

— *Part 13: Intellectual Property Management and Protection (IPMP) extensions*

— *Part 14: MP4 file format*

— *Part 15: Advanced Video Coding (AVC) file format*

— *Part 16: Animation Framework eXtension (AFX)*

— *Part 17: Streaming text format*

— *Part 18: Font compression and streaming*

— *Part 19: Synthesized texture stream*

Introduction

ISO/IEC 14496 specifies a system for the communication of interactive audio-visual scenes. The specification includes the following elements:

1. the coded representation of natural or synthetic, two-dimensional (2D) or three-dimensional (3D) objects that can be manifested audibly and/or visually (audio-visual objects) (specified in part 1,2 and 3 of ISO/IEC 14496);

2. the coded representation of the spatio-temporal positioning of audio-visual objects as well as their behavior in response to interaction (scene description, specified in part 11 of ISO/IEC 14496);

3. the coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content, specified in part 11 of ISO/IEC 14496);

4. a generic interface to the data stream delivery layer functionality (specified in part 6 of ISO/IEC 14496);

5. an application engine for programmatic control of the player: format, delivery of downloadable Java byte code as well as its execution lifecycle and behavior through APIs (specified in part 11 of ISO/IEC 14496); and

6. a file format to contain the media information of an ISO/IEC 14496 presentation in a flexible, extensible format to facilitate interchange, management, editing, and presentation of the media.

The information representation, specified in ISO/IEC 14496-1 and in ISO/IEC 14496-11, describes the means to create an interactive audio-visual scene in terms of coded audio-visual information and associated scene description information. The encoded content is presented to a terminal as the collection of elementary streams. Elementary streams contain the coded representation of either audio or visual data or scene description information or user interaction data. Elementary streams may as well themselves convey information to identify streams, to describe logical dependencies between streams, or to describe information related to the content of the streams. Each elementary stream contains only one type of data.

Elementary streams are decoded using their respective stream-specific decoders. The audio-visual objects are composed according to the scene description information and presented by the terminal's presentation device(s). All these processes are synchronized according to the systems decoder model (SDM) using the synchronization information provided at the synchronization layer.

The scene description stream identifies different types of objects, such as audio, visual, 2D and 3D graphics, etc. that define a scene composition of the content. Synthesized Textures streams provide for photo-realistic animations that can be transmitted using very low bitrates. These type of aniumamtions can be used in combination with other streams to enhance any scene.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of patents.

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured the ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with the ISO and IEC. Information may be obtained from:

Vimatix Inc.
5 Oppenheimer St.
Rehovot 76701
Israel

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Coding of audio-visual objects —

Part 19: **Synthesized texture stream**

1 Scope

This part of ISO/IEC 14496 specifies functionalities for the transmission of Synthesized Texture data as part of the MPEG-4 encoded audio-visual presentation. More specifically, it defines:

1. The synthesized texture format representation that is utilized for Synthesized Texture data encoding
2. The coded representation of Synthesized Texture data streams.

2 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14496-1, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-11, *Information technology — Coding of audio-visual objects — Part 11: Scene description and application engine*

3 Synthesized Texture Compression Technology

3.1 Functionality and Semantics

3.1.1 Overview

Synthesized Textures represent photo-realistic textures by describing color information of vectors. Synthesized Texture streams are used for creation of very low bit rate synthetic video clips. Synthesized Texture clips are built using key frame based animations of skeletons that affect photorealistic textures whose color information is modeled by equations.

A texture top-level **Synthesized Texture** Node (**STNode**) can be defined for playing **SynthesizedTextures,** see ISO/IEC 14496-11 for additional details . The **STNode** itself is similar to the **MovieTexture**, and uses url field to reference an Object Descriptor describing the associated stream(s). The stream contains both the object textures and their animation descriptions . The **STNode** also exposes control points that can be used to manipulate via affine transforms the objects carried in its associated stream. By this way **STNode** can implement synthesized interactive SynthesizedTextures. As any texture, the resulting texture can be mapped onto any 2D or 3D surface.

3.1.1.1 SynthesizedTexture Elements

The **SynthesizedTexture** is a collection of animated **Objects** (also called **Actor**s) sharing a common Stage, Camera and Timeline.

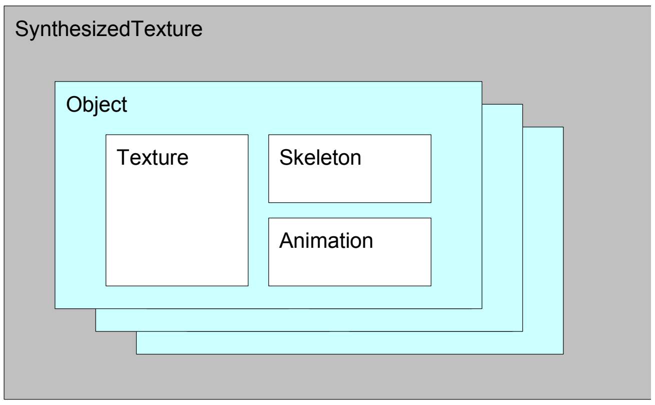


Figure 1 — Synthesized Texture structure

The **Object** is comprised of a Texture, A Skeleton and an Animation.

- The object's **Texture** represents the objects skin.

  The texture is comprised of primitive vector-style entities, belonging to a small number of primitive types such as Lines and Area Color Points.

  The pixel representation of the texture is reconstructed through the process of **Texture Rendering**.

  The texture is divided into mutually exclusive sub-textures called **Layers**.

- The **Skeleton** represents the kinematic capabilities of the object relative to itself and controls the shape and appearance of the skin.

  The skeleton is comprised of a topology of **Bones** whose geometric configuration is controlled by the object's animation.

  The skeleton is attached to the texture's layers, and controls their position and shape within the object's plane. This ultimately affects the layout of the texture primitives on the plain, as the skeleton geometry changes.

  Re-rendering the texture based on a new layout of the texture primitives eventually results in a realistic warping effect called **Texture Warping**.

- The **Animation** represents the spatial behavior of a single object along time.

  The animation of Objects is formed by an extrinsic motion of the entire object relative to the world, and an intrinsic motion of Layers relative to the object they are part of.

The intrinsic motion is controlled by the Skeleton geometry, as described above. Extrinsic motion of each object is controlled by its 3D displacement and rotation within the SynthesizedTexture's world.

The Animation is a sequence of **KeyFrames** describing the state of the object in both intrinsic and extrinsic aspects, at specific frames on a timeline. Frames that are not explicitly described by a key-frame are derived by interpolation between neighboring key-frames, in a process called "tweening".

3.1.1.2 SynthesizedTexture Playback

SynthesizedTexture playback is based on animating all the objects in the SynthesizedTexture as described above.

All objects are animated:

- within the SynthesizedTexture's shared 3D world ("**Stage**"),
- across the SynthesizedTexture's shared **Timeline**,
- and optionally also relative to a camera.

The resulting bitmaps rendered from the objects' Textures are projected onto a shared 2D frame buffer, which holds the current raster frame that is ready for display.

3.1.1.3 SynthesizedTexture Coding

The Synthesized Texture **Bitstream** contains the SynthesizedTexture data in a coded form called **ST Coding**.

ST Coding encodes the SynthesizedTexture's primitive entities in a hierarchical compact arrangement resulting in very high compression rates. This is done by employing several techniques:

a) **Aggregation** – this technique groups and orders specific attributes of primitives into sub-streams, in a manner that is 'friendly' for quantization and packing techniques. For example, location data of all Texture Terminal Points (TPs) and Patches (PAs), are aggregated, ordered according to their "geographic" location, quantized and packed - together. During decoding, sub-stream data are de-multiplexed to their respective primitive types and properties, according to hard-coded rules and soft-coded indicators.

b) **Quantization** – the numeric values in a sub-stream are rounded, factored, offset or otherwise transformed, focused on reducing stream size, and generating data that is 'friendly' to additional packing techniques. Quantization parameters are typically stored in standard hard-coded quantization tables, so that they do not need to be carried in the bitstream. During decoding, quantized sub-streams are de-quantized using the proper quantization tables, through dequant() methods.

c) **Packing** – various loss-less compression[1)] techniques are applied to further reduce data size. These techniques include variants of Huffman, Run-Length, and other encoding methods. During decoding, sub-streams that were packed are un-packed and decompressed, using the appropriate method.

The ST Bitstream includes four types of top-level blocks: Header, Scene, Objects and Textures. The Object block also contains the Skeleton and/or Animation of the Texture.

ST Coding, including its division to sub-streams, their order, and the manner in which they are encoded, may come in one of several **coding syntaxes**. This specification describes the current default syntax called **coding syntax 0**.

1) The terms ***compression*** and ***decompression*** in this document refer to the lossless statistical compression techniques, such as Huffman and Run-Length encoding, used as part of the Synthesized Texture encoding.

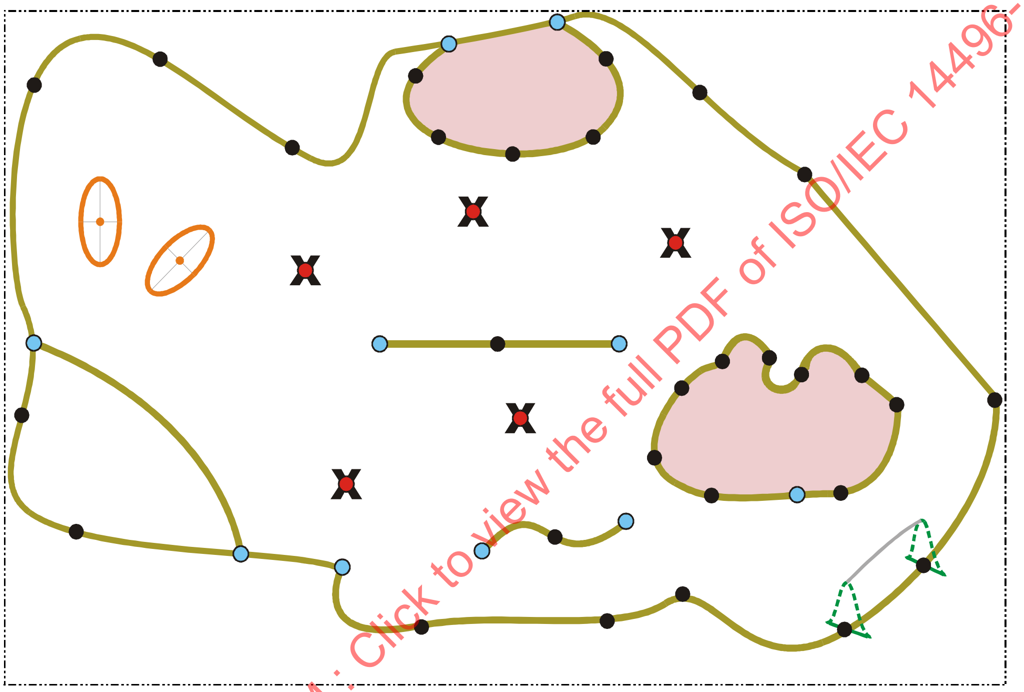
Chapter 4 specifies the structure of the ST Bitstream and describes how it is decoded to SynthesizedTexture primitives.

3.1.2 The Texture

3.1.2.1 General

The Texture is used to represent the skin of an object.

The texture is comprised of small set of primitive vector-style entities.



●━━●	Line (**LN**), bounded by 2 Terminal Points (**TP**)
●━━●	Line Segment (**LS**), bounded by 2 Line Points (**LP**)
⌒	Line Color Profile (**LC**)
●	Area Color Point (**AC**)
━	Patch (**PA**)
-·-·-·-	bounding rectangle

Figure 2 — Synthesized Texture Primitives

3.1.2.2 Primitives and Properties

A Texture is comprised of the following primitive types:

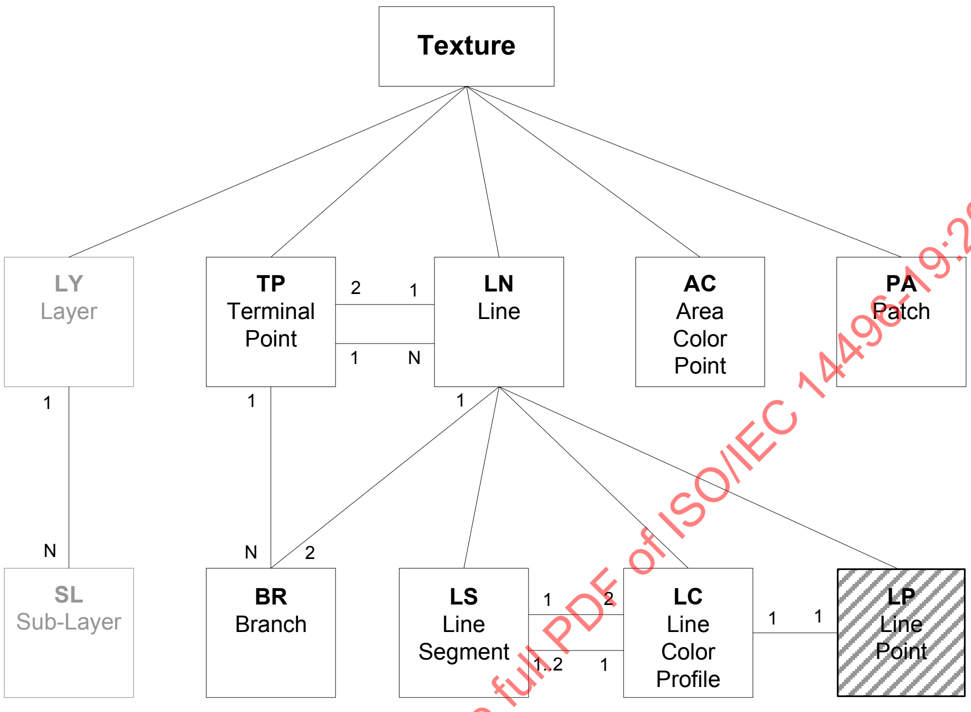


Figure 3 — Synthesized Texture hierarchy of primitives

3.1.2.3 Texture

3.1.2.3.1 Syntax

```
class Texture
{
  int  width, height;

  // decoded texture primitives:
  LN  aLN[];          // Lines
  TP  aTP[];          // Terminal Points
  AC  aAC[];          // Area Color Points
  PA  aPA[];          // Patches

  int nLN = 0;        // Number of LNs
  int nTP = 0;        // Number of TPs
  int nAC = 0;        // Number of ACs
  int nPA = 0;        // Number of PAs

  int   worldUnit = 512;
  float LosOffsetX = 0;
  float LosOffsetY = 0;

  // auxiliary
  LY  aLY[];        // Layers
  int nLY = 0;      // Number of LYs
  int nSL = 0;      // total number of sub-layers in texture
}
```

3.1.2.3.2 Semantics

Properties:

Name	Description
width, height	The width and height of the texture's bounding rectangle, in pixels. All coordinates of the texture's elements are given relative to the top-left corner of this rectangle.
Ordered arrays of decoded texture primitives:	
aLN	The Lines in the Texture
aTP	The Terminal Points in the Texture
aAC	The Area Color Points in the Texture
aPA	The Patches in the Texture
nLN	The number of elements in aLN
nTP	The number of elements in aTP
nAC	The number of elements in aAC
nPA	The number of elements in aPA
worldUnit	The number of pixels corresponding to 1 "world-unit" in this texture. World units are used to describe distances in the SynthesizedTexture's 3D world. Default: worldUnit = 512 pixels.
LosOffsetX, LosOffsetY	The x,y components of the texture's LOS (Line of Sight) Offset, in pixels. A texture's LOS Offset is the offset of the texture's center from the center of the visual plane of a camera that could have captured the texture. This information allows adapting the appearance of a texture which was photographed from a certain angle (relative to the camera's LOS), to the changing angle in which it is viewed in an animated SynthesizedTexture. For a texture that was photographed from a "straight ahead" angle, the LOS Offset is (0, 0).
Auxiliary:	
aLY	The Layers in the Texture Note layers are an auxiliary structure not used in Texture Rendering; they serve Texture Decoding only for associating Texture primitives to bones in the skeleton.
nLY	The number of elements aLY
nSL	The total number of sub-layers in the Layers in the texture

3.1.2.4 Line (LN)

3.1.2.4.1 Syntax

```
class LN      //  Line
{
  int iLnType;
  int iContourType = 0;
  int iSlLft, iSlRgt;

  int iTpBeg, iTpEnd;

  LS  aLS[];
  int nLS;

  LC  aLC[];
  int nLC;
}
```

3.1.2.4.2 Semantics

Lines implement the **characteristic lines** of a Texture. A Line has a geometry described by Line Segments (LSs) and a coloring described by Line Color Profiles (LCs) placed along its course.

A Line primitive is built of a chain of Line Segment (LSs), and its 2D geometry is determined by these segments. Every Line Segment is part of exactly one single Line, and is terminated by exactly 2 Line Points (LPs) – one at each end. In addition to the LPs at the 2 ends of each line, a Line is always terminated by exactly 2 Terminal Points (TP) – one at each end.

Every Line Point hosts exactly 1 Line Color Profile (LC), and every LC is currently hosted by exactly 1 LP. For a closed Line, the number of LSs equals the number of LPs & LCs; For a non-closed Line LSs are 1 less than LPs & LCs.

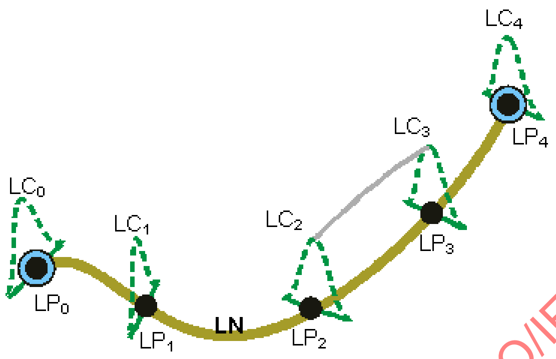


Figure 4 — Synthesized Texture Line (LN)

The collection of LCs along a Line determines the coloring behavior of the line, together with the Line's iLnType.

The external contour of a Texture, and borders between Layers and Sub-Layers are built of Lines. Lines participating in a contour are marked by using the Line's iContourType.

Properties:

Name	**Description**
iLnType	The type of coloring behavior across the "width" dimension of this Line. This indicates the type of Line Color Profiles (LCs, §3.1.2.8) used along the Line. 00-RIDGE. 01-STRIPE. 02-EDGE. 03-ABSENT (Absent out-branch in Splitting). 04-PARALLEL (Parallel Ridge linked to out-branch in Splitting). A distinction is made between separating and non-separating line types. Separating lines block the "diffusion" of color originating at Area Color Points (ACs) near the Line from one side of the Line to the other. RIDGE and EDGE lines are separating, while STRIPE lines are non-separating.
iContourType	Marks whether this Line is part of the contour (i.e. outer border) of the Texture or one of its layers. 00-This LIne is not a contour line.
iSlLft, iSlRgt	When this Line separates Sub-layers – the indexes of the sub-layers to the left and to the right of the Line.
iTpBeg, iTpEnd	The indexes of the TPs in which this Line begins and ends.
aLS	The array of Line Segments comprising the geometry of this Line.
nLS	The number of elements in aLS. includes last LS ?
aLC	The array of Line Color Profiles describing the coloring of this Line.
nLC	The number of elements in aLC. nLC = nLS+1.

3.1.2.5 Line Segment (LS)

3.1.2.5.1 Syntax

```
class LS                    // Line Segment Geometry
{
  float vX, vY;
  float hgt;
}
```

3.1.2.5.2 Semantics

Line Segments are the building blocks of Lines (LNs).

Every Line Segment is part of exactly one Line, and is terminated by exactly 2 Line Points (LPs) – one at each end.

A Line Segment is an arc which follows a globally fixed mathematical model. In the current implementation of the Texture, a Line Segment is unambiguously represented as a parabolic curve. The LS parabola geometry is encoded using its vX, vY and hgt.

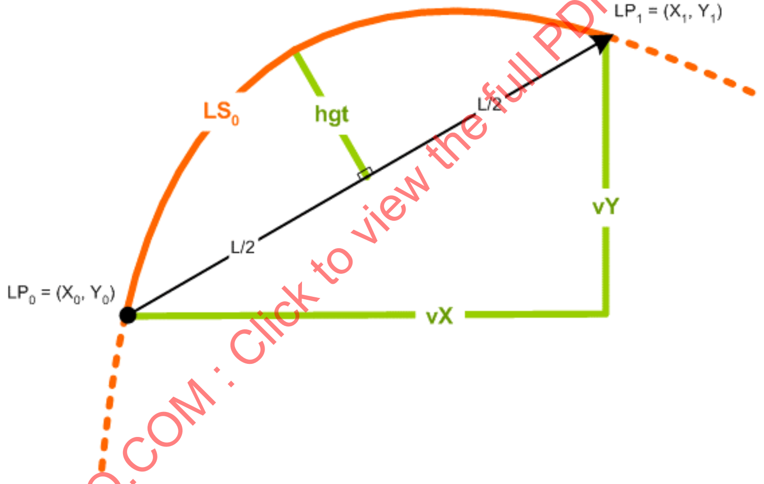


Figure 5 — Synthesized Texture Line Segment (LS)

The base of the LS arc is described by a vector, given by the LS's starting point LP0 and the (x,y) components of the vector – vX and vY.

The signed height of the LS arc above or below its base is given by LS.hgt.

The LS parabola is defined only between the LS's starting point LP0 and its end point LP1.

Properties:

Name	**Description**
vX, vY	The x,y components of the vector connecting this Line Segment's starting and ending points, in pixel units. The LS curve is defined only between the LS's starting and ending points (LP0 and LP1).
hgt	The signed height of the parabola of this LS above or below the LS's vector.

3.1.2.6 Line Points (LP)

3.1.2.6.1 Syntax

```
class LP                    // Line Point
{
}
```

3.1.2.6.2 Semantics

A **Line Point** indicates the location of the Line Segments (LSs), and Line Color Profiles (LCs) in the texture.

A Line Point is the meeting point between 2 adjacent Line Segments, i.e. every LS connects exactly 2 LPs. Every LC in the texture resides on a single corresponding LP.

In practice, the LP entity is not represented explicitly in the Texture data structures and bitstream, but it is mentioned in the primitive model for completeness. The LP location information can be derived from the LN's starting and ending Terminal Points and its Line Segments.

3.1.2.7 Terminal Point (TP)

3.1.2.7.1 Syntax

```
class TP                    // Terminal Point
{
  float x, y;

  BR   aBR[];               // branches from this TP
  int  nBR;                 // number of branches from this TP
  int  junctionType;

  class BR
  {
    int    iLN;             // index of LN that branches from this BR
    bool   bOut;            // is this branch an "out-branch"?
    int    iLnType;         // Line Type of the branching LN.
  }
}
```

3.1.2.7.2 Semantics

Terminal Points are a higher level structure, providing additional information regarding the coloring behavior at crossings, splittings and end points of lines. Every "Line" is terminated by exactly 2 TPs – one at each end.

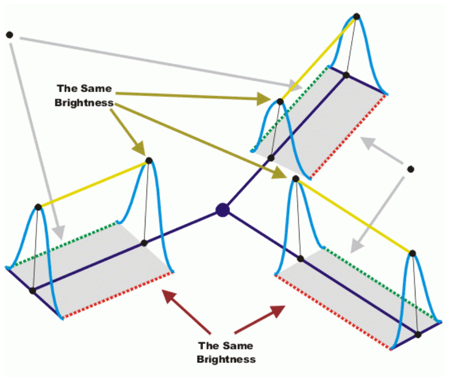


Figure 6 — Synthesized Texture Terminal Points (TP)

Terminal Points occur in the following circumstances:

- At any intersection in which 3 or more Lines are branching out – a TP is mandatory.
- At the end of a Line, where the Line does not connect to any other Line – a TP is mandatory.
- On a closed Line, (i.e. where the Line has no obvious end point), at least one TP must exist at an arbitrary point on the Line.
- At any point on a line, thus splitting that line into 2 "Lines".

Properties:

Name	**Description**
x, y	The coordinates of this TP in the Texture.
ABR	The array of branches from this TP.
NBR	The number of branches in aBR.
junctionType	The junction type of this TP: 0 - ALLRIDGE junction (separating). 1 - ALLSTRIPE junction (STRIPE = non-separating ridge). 2 - ALLEDGE junction (separating) 3 - SPLITTING junction (1 RIDGE in and 0 or 1 EDGEs out)
class BR: a branch in which LNs go "in" or "out" of a TP.	
iLN	The index of the Line that branches from this BR.
bOut	Is this branch an "out-branch"? "out -branches" and "in -branches" are branches in which a Line "starts" and "ends" respectively. "in" and "out" is an arbitrary encoding decision, but every Line starts in an out-branch and ends in an in-branch.
iLnType	The line type of the branching Line. See LN.iLnType.

3.1.2.8 Line Color Profile (LC)

3.1.2.8.1 Syntax

```
class LC                                // Line Color Profiles
{
  float depth = 0;
}
```

```
class LCseparating                      // Separating (EDGE or RIDGE) Line Color Profile
{
  color colorFarLft, colorFarRgt;
  color colorMidLft, colorMidRgt;
}
```

```
class LCedge                            // EDGE Line Color Profile
extends LCseparating
{
  float width;
}
```

```
class LCridge                            // RIDGE Line Color Profile
extends LCseparating
{
  color colorCenter;
  float widthLft;
  float widthRgt;
}
```

```
class LCstripe                        // STRIPE Line Color Profile
extends LC
{
  color colorCenter;
  color colorCurvature;
}
```

3.1.2.8.2 Semantics

Line Color Profiles (LCs) describe the coloring behavior along Lines.

Each colorXxx parameter describes the brightness of each of the 3 color components (Y, Cr, Cb) at a certain distance (center, middle, or far), and side (left or right) relative to the profile's geometrical center.

Line Color Profiles have 3 morphologies, corresponding to the 3 possible Line types: EDGE, RIDGE and STRIPE. All LCs on a Line are of the same type, given in the Line's iLnType.

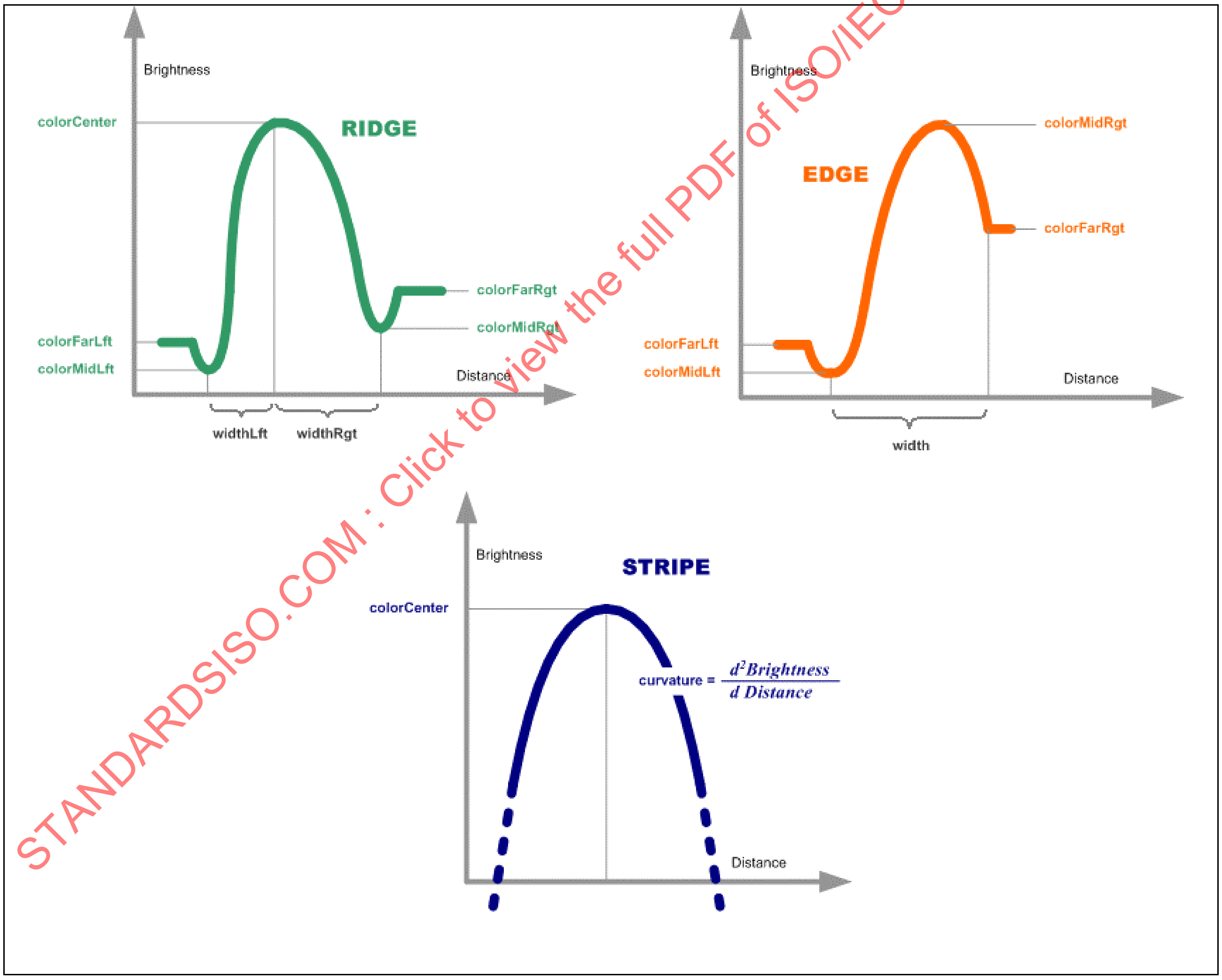


Figure 7 — Synthesized Texture Color Profiles (CP)

In the current implementation of SynthesizedTexture, an LC is provided for every location of a Line Point (LP) in the Texture, so its location is given by the corresponding LP's location[2].

A LN includes an array of LCs (aLC[nLC]) describing its coloring behavior, where nLC = nLS + 1. Thus for a Line with nLS segments, the location of LC_i for i=0..nLS-2 is the location (LS_i.x, LS_i.y), and the location of the last LC on the Line (i = nLS-1) is ($TP_{LSi.iTpEnd.x}$, $TP_{LSi.iTpEnd.y}$).

For convenience, the LC is also used to store the "depth" value for its respective Line Point.

Properties:

Name	**Description**
LC	Line Color Profile base class
Depth	The depth of this LC's Line at the location of this LC. Default=0.0.
LCseparating	Separating (EDGE or RIDGE) Line Color Profile base class
colorFarLft, colorFarRgt	The far-left and far-right colors of this LC.
colorMidLft, colorMidRgt	The mid-left and mid-right colors of this LC.
Lcridge	RIDGE Line Color Profile
ColorCenter	The central color of this LC.
widthLft, widthRgt	The left and right widths of this LC's Line at the location of this LC, in pixels.
Lcedge	EDGE Line Color Profile
Width	The width of this LC's Line at the location of this LC, in pixels.
Lcstripe	STRIPE Line Color Profile
ColorCenter	The central color of this LC.
ColorCurvature	The "color-curvature" of this stripe. If this stripe is described by a parabola $y=Ax^2+B$, then the color-curvature is the coefficient A.

3.1.2.9 Patch (PA)

3.1.2.9.1 Syntax

```
class PA
{
  float x, y;
  float r1, r2;
  float ang;

  color colorCenter;        // central color

  int iSL;

  float depth = 0;       // depth of this PA; may be absent.
}
```

2) This arrangement is economical as it saves the space needed to store LC location information, and does not come from deeper imaging grounds; note that LCs could theoretically be located on LNs according to an alternative scheme.

3.1.2.9.2 Semantics

A **Patch** is a small ellipse, typically a few pixels long, whose color is significantly different from its surrounding "area color".

The geometry of a Patch is described by the ellipse's center point, primary and secondary radius, and the angle of its primary radius relative to the texture's coordinate system.

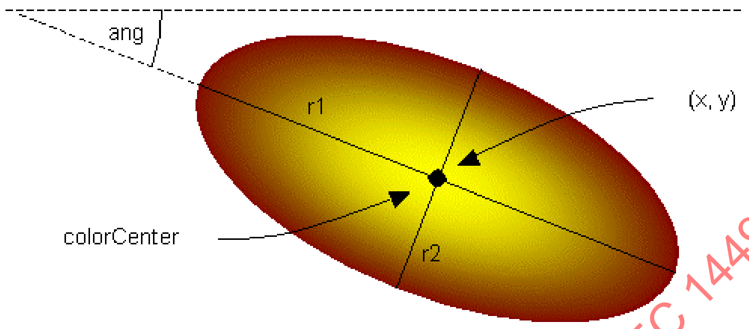


Figure 8 — Synthesized Texture Patch (PA)

The coloring of the Patch is described by the color at its geometric center, which can then be blended with the area color at its perimeter.

Properties:

Name	Description
x, y	The x and y coordinates of the geometrical center of the ellipse which represents this Patch.
R1, r2	The length of the primary and secondary radius respectively of the ellipse which represents this Patch, in pixels.
Ang	The angle of r1 relative to the Texture's X axis, in whole degrees, 0°..359°.
colorCenter	The central color of this PA.
ISL	The index of the sub-layer this PA is on.
Depth	The depth of this PA. Default=0.0.

3.1.2.10 Area Color Point (AC)

3.1.2.10.1 Syntax

```
class AC                            // Area Color Points
{
  float x, y;

  color colorCenter;     // central color

  float depth = 0;

  int iSL
}
```

3.1.2.10.2 Semantics

Area Color Points (ACs) describe the low-scale color changes in the areas between Lines, and more specifically between "separating" Lines, i.e. LNs of LineType EDGE and RIDGE.

Properties:

Name	Description
x, y	The coordinates of this AC in the Texture.
colorCenter	The color at the center of the AC.
depth	The depth of this AC. Default=0.0.
iSL	The index of the sub-layer this AC is on.

3.1.2.11 Layer (LY)

3.1.2.11.1 Syntax

```
class LY
{
  char  name[64];

  int   X0, X1, Y0, Y1;   // coords of LY's bounding rect
  bool  bHasDepthDeltas;

  SL    aSL[];
  int   nSL = 0;

  class SL
  {
    point3D orient;
    float   dist;
    bool    bOrthogonal;
    int     surfaceType;
    int     iLY;
  }
}
```

3.1.2.11.2 Semantics

Layers and **Sub-Layers** are auxiliary structures used in SynthesizedTexture authoring and encoding to group Texture primitives. Properties such as Depth and "controlling-polybone" are then attributed in bulk to all primitives in the Layer or Sub-Layer.

Properties:

Name	Description
name	The name of the layer, given by the author, used as reference in composition applications.
X0, X1, Y0, Y1	The coordinates of the layer's bounding rectangle.
bHasDepthDeltas	Do texture primitives in this layer have depth deltas (otherwise they inherit their depth from their position on the layer plane). Relevant only if decHeader.bHasDepthDeltas is true;
aSL[]	An array of sub-layers that are part of this LY.
nSL	Number of sub-layers in this LY.
class SL - sub-layer:	
orient	The normalized (x,y,z) representation of this sub-layer's orientation.
dist	The distance of this sub-layer in worldUnits. Typical 1..1000.
bOrthogonal	Is this sub-layer orthogonal to the line of sight? If true then orient is (0, 0, 1.0).
surfaceType	The type of geometry of this sub-layer. default: 0=PLANE.
iLY	Cross reference to the the index of the LY that this sub-layer is in.

3.1.2.12 Color

3.1.2.12.1 Syntax

```
class color
{
  int y, cr, cb;
  int R, G, B;
}
```

```
color::toRGB()
{
  R = y + 1.40200*cr
  G = y - 0.34414*cb - 0.71414*cr
  B = y + 1.77200*cb
}
```

3.1.2.12.2 Semantics

A **color** object can describe color in two methods: YCrCb and RGB.

YCrCb color values are used for encoding colors in the SynthesizedTexture bitstream, as they display better aggregation and compression behavior.

In the decoding process, YCrCb color values are converted to RGB colors, which are then used in Texture rendering.

3.1.3 Texture Rendering

3.1.3.1 General

The process of Texture Rendering reconstructs a displayable bitmap image from SynthesizedTexture primitives.

The reconstruction algorithm starts with the input parameters and computes the brightness (color) value of the synthesized image at each of its pixels. It consists of the following principal parts:

Computing brightness of Lines (Procedure BL below).

Computing brightness of Patches (Procedure BP below).

Computing Area coloring, i.e. brightness of the "Background", area between Lines and Patches (Procedure BB below).

Blending the computed brightness values into the final image (The MAIN Procedure below)

In the case of a color image these computations are performed independently for each color component.

3.1.3.2 Procedure MAIN: Computing Final Brightness Values

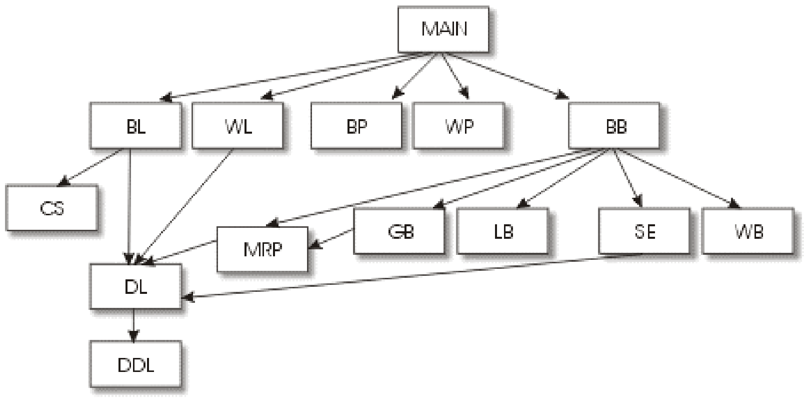


Figure 9 — Synthesized Texture Rendering Flow

For any point z in the image plane the final brightness B(z) of the Synthesized image at the point z is computed according to the following formula:

$$B(z) = \frac{WL(z)\cdot BL(z) + WB(z)\cdot BB(z) + \sum_s WP_s(z)\cdot BP_s(z)}{WL(z) + WB(z) + \sum_s WP_s(z)} \qquad (1)$$

Here *BB(z)*, *BL(z)* and *BP$_s$(z)* are the brightness functions of the Background, of the Lines and of the Patches, computed in the Procedures **BB**, **BL** and **BP**, respectively, and the sum $\sum_s$ runs over all the Patches P_s.

The weight functions *WL(z)* and *WP$_s$(z)* are computed in the Procedures **WL** and **WP**, respectively, and

$WB(z) = 1 - max(WL(z), WP_s(z))$.

Division by the sum of all weight functions in formula (1) guarantees that their sum is identically 1 and that formula (1) is a normalized averaging.

3.1.3.3 Procedure BL: Brightness due to Lines

This procedure computes for any point z on the image the brightness BL(z), contributed by the Line Color Profiles (LCs) of the Lines (LNs) in the texture.

BL(z) needs to be computed only for those z which are "close enough" to at least one of the Lines in the texture, as expressed by the weight function WL.

Note: The most intuitive and natural way to define the brightness of a Line is to associate to it a coordinate system (uu, tt), with uu(z) the (signed) distance of the point z to the line along the normal direction, and tt(z) the length parameter along the curve of the orthogonal projection pp(z) of z onto the line.

Then the brightness cross-sections are computed according to the coordinate uu and interpolated along the line with respect to the coordinate tt.

The corresponding algorithm can be constructed. However, it provides some serious drawbacks:

- Actual computing of the coordinates uu and tt is a mathematically complicated task.

- Even for smooth curves without corners the normal direction is correctly defined only in a small neighborhood of the curve (of the size of a couple of pixels in realistic situations). Outside this neighborhood no natural mathematical solution exist for defining the normal, the projection etc.

- For spline curves with corners between some of their links (which usually appear in realistic situations) the normal is not defined even locally. Once more, the situation can be corrected by using the mid of the corner angles, but the global difficulties of (2) remain and algorithms become rather complex.

Consequently, we show below an algorithm, which can be considered as an approximation to the “ideal one” above. Its main advantage is that the “coordinates” uu and tt (called below u and t) can be computed independently for each Line Segment (link) of all the collection of Lines. Moreover, the computation can be ultimately rendered as rather efficient (although the description below may look somewhat complicated).

Below for any point z, u(z) is the “distance of z to Lines”, S(z) is the closest link to z (with respect to the distance u) in the collection of Lines, and t(z) is the parameter, measuring the projection of z onto S(z), rescaled to the segment [0, 1]. S(z), u(z) and t(z) are computed by the Procedure **DL**, described below.

The Procedure BL distinguished whether the Line Segment S(z) has a “free end” (i.e. an endpoint TP, not belonging to any other Line Segment) or not:

1. S(z) does not have “free ends”.

Let C_1 and C_2 denote the equations of the two cross-sections (normalized to the unit width, as described in the Procedure **CS** below) at the two endpoints of the link S(z). For u(z) > 0 let W_1 and W_2 denote the respective right widths RW_1 and RW_2 of the cross-sections at these points. For u(z) < 0 let W_1 and W_2 denote the respective left widths LW_1 and LW_2 of the cross-sections at these points. Then in each case

$$BL(z) = t(z) \cdot C_1\left(\frac{u(z)}{W(z)}\right) + (1 - t(z)) \cdot C_2\left(\frac{u(z)}{W(z)}\right)$$

where W(z) is the interpolated width

$$W(z) = t(z) \cdot W_1 + (1 - t(z)) \cdot W_2$$

and the values $C_1\left(\frac{u(z)}{W(z)}\right)$ and $C_2\left(\frac{u(z)}{W(z)}\right)$ are computed by the procedure CS.

S(z) has a “free end”.

If for this “free end” the parameter t is zero, the brightness BL(z) is computed as above for t(z) > 0. For $-DE < t(z) < 0$,

$$BL(z) = [1 + t(z)/DE] \cdot C_1\left(\frac{u(z)}{W(z)}\right) - [t(z)/DE] \cdot BM,$$

and for $t(z) < -DE$,

$$BL(z) = BM.$$

Here DE is a positive tuning parameter, defining the shape of the end of a Line. BM is half of the sum of the brightness parameters colorMidLft and colorMidRgt of the cross-section at the free end.

If for this “free end” the parameter t is one, t(z) is replaced by 1-t(z) in the above formula.

Note: The formula above provides one of possible choices of the shape of Lines near their ends. It assumes that the cross-section brightness gradually descends to the “mid value” value BM inside the prescribed distance DE. Other shapes can be defined, by properly computing the width and the brightness in the neighborhood of the end point.

3.1.3.4 Procedure CS: Color Profiles (Cross-Sections) of Lines

This procedure computes a brightness value of an edge or a ridge (unit width) cross-section CS(u) for any given cross-section “interior” brightness parameters, as described above, and for any value of u.

In the Procedure BL u is the distance u(z) to the line, normalized by the width W(z) of the line, so the width parameter W is taken into account inside the BL, and it does not appear below. Similarly, the margin brightness parameters colorFarLft and colorFarRgt enter the computations in the Background brightness Procedure BB.

3.1.3.4.1 Edge Cross-Section.

Normalized edge cross-section NEC(u) is defined as follows (Figure 10 — Edge cross-section definition.):

$$NEC(u) = \begin{cases} u < -1, & 0 \\ u > 1, & 1 \\ -1 < u < 0, & 0.5(u+1)^2 \\ 0 < u < 1, & 1 - 0.5(u-1)^2 \end{cases}$$

Thus the recommended edge cross-section is composed of two symmetric parabolic segments.

For given brightness parameters colorMidLft and colorMidRgt, the value CS(u) is computed as

$$CS(u) = LB2 + (RB2 - LB2) \cdot NEC(u)$$

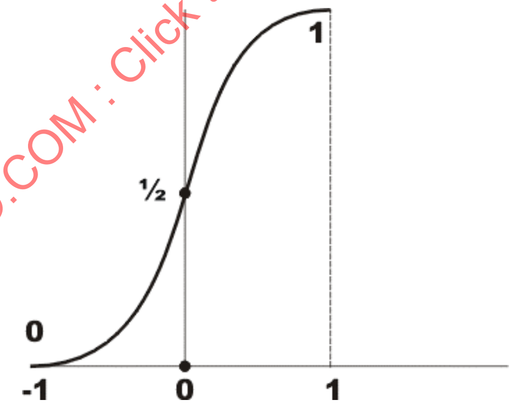


Figure 10 — Edge cross-section definition.

3.1.3.4.2 Ridge Cross-Section

As for edges, the width of the ridges is taken into account in the Procedure BL. Similarly, the margin brightness parameters colorFarLft and colorFarRgt enter the computations in the Background brightness Procedure BB. Consequently the ridge cross-section computed in the current Procedure CS, is the same for separating and non-separating ridges, and is defined by the parameters colorMidLft, colorCenter and colorMidRgt, as follows (Figure 11 — Ridge cross-section definition.

$CS(u) = LB2 + (CB - LB2) \cdot NEC(2u+1),$ for $u < 0$, and

$$CS(u) = RB2 + (CB - RB2) \cdot NEC(-2u + 1), \quad \text{for } u > 0.$$

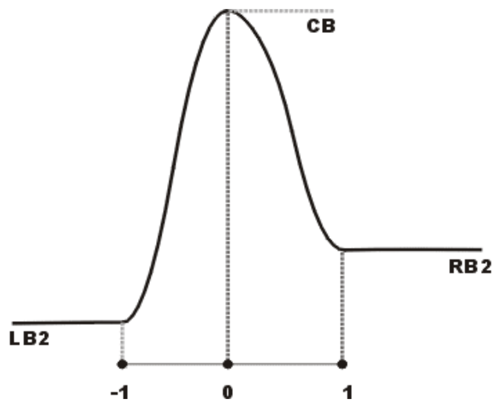


Figure 11 — Ridge cross-section definition.

Thus the recommended ridge cross-section is composed of two edge cross-sections, properly aggregated.

In the process of the blending of these cross-sections with the Background (which incorporates the margin brightness values colorFarLft and colorFarRgt) we get back essentially the same cross-section, as shown below (Figure 12 — Blending two cross-sections with the Background.).

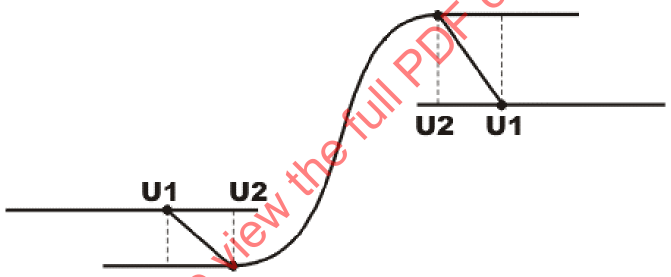


Figure 12 — Blending two cross-sections with the Background.

3.1.3.5 Procedure WL: Weight Function of Lines

This block computes the weight function WL(z), which is used in a final blending of the Lines with the Background. The function WL(z) is equal to one in a certain neighborhood of the Lines, and is zero outside of a certain larger neighborhood.

More accurately:

$$WL(z) = \begin{cases} 1 & |u(z)| < UL_2 \cdot W(z) \\ \dfrac{W(z) \cdot UL_1 - |u(z)|}{W(z) \cdot (UL_1 - UL_2)} & UL_2 \cdot W(z) < |u(z)| < UL_1 \cdot W(z) \\ 0 & |u(z)| > UL_1 \cdot W(z) \end{cases}$$

The distance u(z) is computed in the Procedure **DL**.

UL_1 and UL_2 are tuning parameters; see the last section “Tuning Parameters”.

Figure 13 shows a typical cross-section and a general shape of the weight function WL(z).

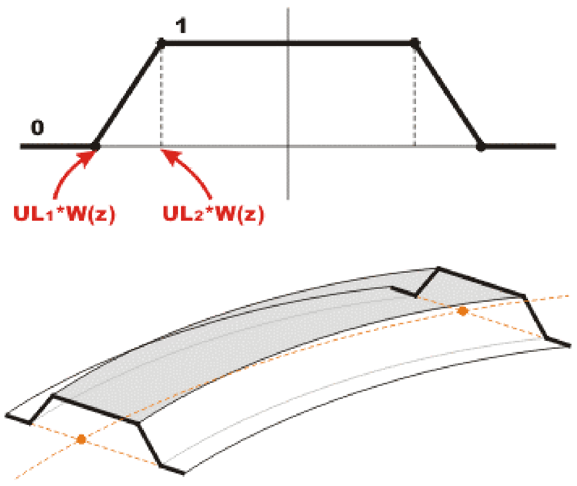


Figure 13 — Typical cross-section and general shape of WL(z)

3.1.3.6 Procedure DL: Distance to Lines

This Procedure computes for any point z in the texture:

1. The point p(z) on the Lines which is nearest to z, i.e. the "projection" p(z) of z onto the set of Lines.

d) The distance u(z) between z and p(z).

e) The link S(z) on which p(z) resides.

f) The proportion t(z) in which p(z) divides the link S(z).

Note u(z), p(z) and t(z) are NOT exactly the Euclidean distance, the corresponding mathematical projection and proportion respectively; however, in most cases they give a reasonable approximation for these mathematical entities.

These data are computed in the following steps:

2. For each link (Line Segment, LS) Si in the texture, the corresponding pi(z), ui(z), ti(z) are computed in Procedure DDL (See the Synthesized-C figure below)

g) S(z) is defined as the link Sj, for which the minimum of the absolute values |ui(z)| is attained (See the figure Synthesized-D below)

h) u(z) is defined as the function uj(z) for the link Sj = S(z)

i) t(z) is defined as tj(z) for the above link Sj

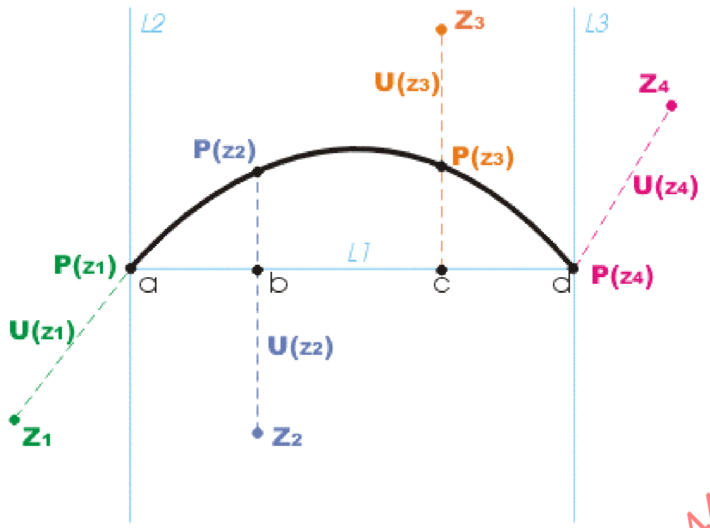


Figure 14 — Synthesized-C

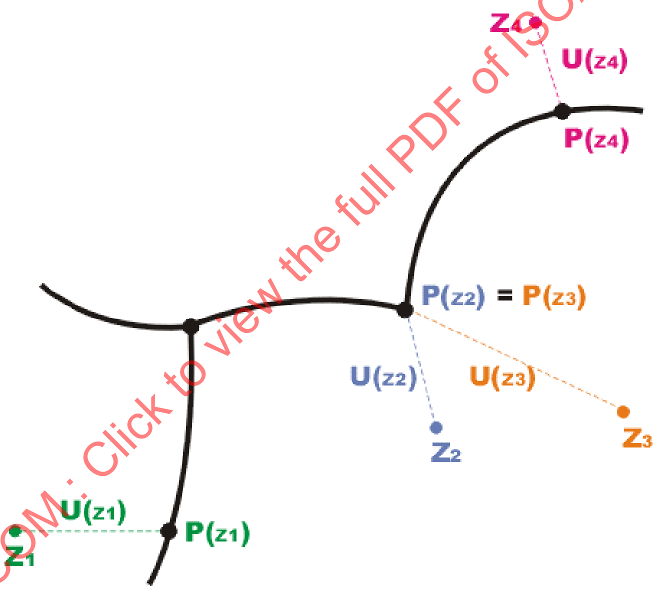


Figure 15 — Synthesized-D

3.1.3.7 Procedure DDL: Distance to a Line Segment

This procedure computes for any point z its (signed) distance u(z) to a given link S (Line Segment, LS), the projection p(z) of the point z onto the link S and the parameter t(z). The Procedure is essentially represented on figure Synthesized-C above (which shows, in particular, equidistant lines for the points z1 and z4.

The straight oriented segment [a, d], joining the end points of the link S is constructed, with the orientation, induced from the orientation of the Line, containing S. l_1 is the straight line, containing the segment [a, d]. l_2 and l_3 are the straight lines, orthogonal to l_1 and passing through a and d, respectively.

Now, for any z in the image plane, the function u(z) is constructed as follows:

For z between l_2 and l_3, the absolute value |u(z)| is the length of the segment, joining z and S and orthogonal to l_1.

For z left to l_2 , |u(z)| is defined as follows:

$$|u(z)| = \left[d(z,l_1)^2 + Dd(z,l_2)^2\right]^{1/2}.$$

Here d(z, l_1) and d(z, l_2) are the distances from z to $l_1$and

l_2 respectively. D is a tuning parameter, with a typical value D = 4.

For z right to l_3 , |u(z)| is defined as

$$|u(z)| = \left[d(z,l_1)^2 + Dd(z,l_3)^2\right]^{1/2}.$$

Let *ll* be an oriented line, formed by the interval of the line l_1 from infinity to a, then by the link S from a to d, and then by the interval of the line l_1 from d to infinity.

For z right to *ll* (with an orientation as above) the sign of u(z) is "+". For z left to *ll*, the sign of u(z) is "-".

For z between l_2 and l_3, the projection p(z) is the intersection point of S and of the segment, joining z and S and orthogonal to l_1 . For z left to l_2 , p(z) is a, and for z right to l_3 , p(z) is d.

For any z, t(z) is the proportion, in which the projection of z onto the line l_1 subdivides the segment [a, d]. For example, for the point z_2 and z_3 on Fig Synthesized-D above. t(z_2) = (b-a)/(d-a), and t(z_3) = (c-a)/(d-a), respectively. For z left to l_2 , t(z) < 0, and for z right to l_3 , t(z) > 1.

Note: The special form of the function u(z) above (for z outside the strip between l_2 and l_3) is motivated by the following reason: when computing in the Procedure BL the brightness of the line near a sharp corner, the form of the distance function u(z) determines which link will be taken as the closest to the points in the sector stressed on the Figure below. For the distance, computed as above, with the parameter D > 1, this choice is matched with the sign of u(z), as defined above. If we would have chosen D < 1, for z in the sector stressed on the figure below, the choice of the nearest link, together with the proposed computation of the sign of u(z), would produce a color from the incorrect side of the line. See the figure below.

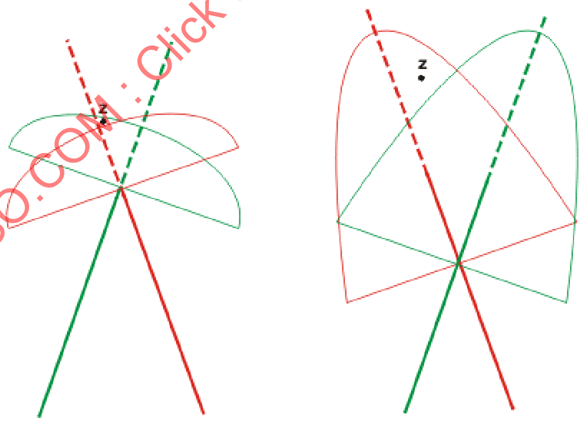


Figure 16 — Distance based selection of line segment

3.1.3.8 Procedure BP: Brightness of Patches

Let Cx, Cy, R1, R2, a, colorCenter and MB be the parameters of a certain Patch P_s, as described above.

Let M be the linear transformation of the plane, transforming the basis ellipse of the Patch to the unit circle. M is a product of the translation by (-Cx, -Cy), the rotation matrix to the angle –a, and the rescaling 1/R1 and 1/R2 times along the x and y axes, respectively. If we put for

$$z = (x, y) , (x'(z), y'(z)) = M(x, y) = M(z),$$

then the equation of the basis ellipse of the Patch is given by

$$x'(z)^2 + y'(z)^2 = 1.$$

The brightness function $BP_s(z)$ of the Patch is then given by

$$BP_s(z) = 0 \text{ for } x'(z)^2 + y'(z)^2 > UP1^2,$$

$$BP_s(z) = MB \text{ for } 1 < x'(z)^2 + y'(z)^2 < UP1^2, \text{ and}$$

$$BP_s(z) = MB + (CB - MB) \cdot (1 - x'(z)^2 - y'(z)^2), \text{ for } x'(z)^2 + y'(z)^2 < 1.$$

Here UP1 > 1 is a parameter. See the Figure below.

3.1.3.9 Procedure WP: Weight Function of Patches

The weight function $WP_s(z)$ for a Patch P_s as above is defined by

$$WP_s(z) = 0 \text{ for } uu(z) > UP1,$$

$$WP_s(z) = 1 \text{ for } uu(z) < UP2, \text{ and}$$

$$WP_s(z) = (UP1 - uu(z)) / (UP1 - UP2)$$

for uu(z) between UP2 and UP1,

where uu(z) denotes the square root of $x'(z)^2 + y'(z)^2$.

Here UP2, 1 < UP2 < UP1, is another tuning parameter. See the figure below.

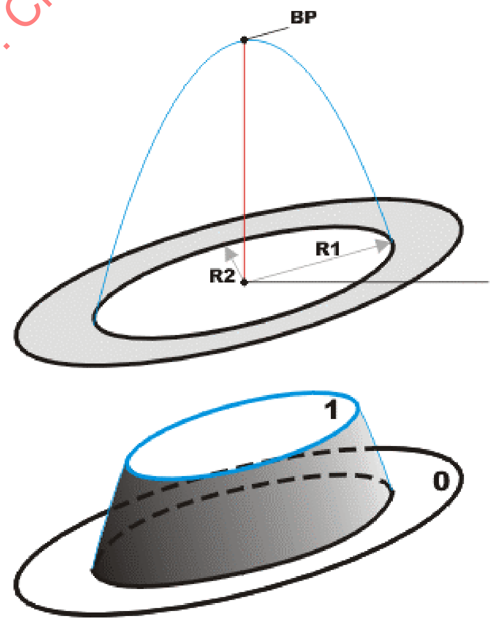


Figure 17 — Synthesized Texture Patch and its weight function

3.1.3.10 Procedure BB: Brightness of Background

This Procedure computes the brightness value of the Background at any point z of the image. This value is obtained as a result of interpolation between the “global” Background brightness values, the margin brightness values of the Lines and the brightness values at the Area Color Points (ACs). The main difficulty is that the interpolation is not allowed to cross the separating lines. To overcome this difficulty a special “distance” d between the points on the image is introduced, which is the length of the shortest pass, joining these points, and not crossing separating Lines, and which is computed in the Procedure **SE** below. Then averaging weights are computed through the distance d.

This block uses as an input a certain collection of the ACs Z_i, (containing the input ACs, as described above, and the margin representing points, produced by the block “**MRP**’, described below). At each point Z_i the brightness value Bb_i is given.

The Background brightness value BB(z) is finally produces by the block BB as follows:

BB(z) is the weighted sum of the global brightness GB and of the Local brightness functions $Bb_i(z)$ over all the ACs Z_i :

$$(2) \qquad BB(z) = (1/S_1(z))[WG(z)BG + \sum_i WR(d(z,Z_i))Bb_i(z)]$$

Here $S_1(z) = WG(z) + \sum_i WR(d(z,Z_i)),$

so the expression (2) is normalized to provide a true averaging of the corresponding partial values.

The global brightness value BG is provided by the Procedure **GB** below, or by any of the Gradient Nodes.

The computation of the Local brightness functions $Bb_i(z)$ is performed in the Procedure **LB** below.

The distance functions d(z, Z_i) are computed in the Procedure **SE** below.

The computation of the weight functions WR(d(z, Z_i)) is performed in the Procedure **WB** below.

The weight GW(z) of the global Background value GB is defined as

$$GW(z) = 1 - \max_i WR(d(z,Z_i)).$$

GW(z) is zero at any z, where at least one of the weights of the representing points is 1, and GW(z) is one at any z where all the weights of the ACs vanish.

3.1.3.11 Procedure GB: Global Brightness of Background

This Procedure computes the global Background value GB, which appears in the expression (2) in the Procedure BB.

By definition, if the point z is inside the Background region of a Sub-Texture number r, for which the global value GBr is defined, GB is equal to this global value GBr. If the point z is inside the Background region of a Sub-Texture, for which the global Background value is not defined, GB is equal to the default global value DGB. If DGB is not defined, GB is equal to zero. Alternatively, Color Gradients can be used.

The current procedure consists in a signal expansion that transmits to each pixel its Sub-Texture number. We describe it shortly, since it essentially belongs to a higher data representation level.

First the procedure MRP is applied, which creates margin representing points, carrying the corresponding Sub-Texture numbers. These numbers are taken from the corresponding poly-links.

Second, the Signal Expansion Procedure is applied to the margin representing points, essentially as in the block SE, with the following difference: only the marking and the number of the Sub-Texture is transmitted between the pixels on each step of signal expansion.

As this procedure is completed, each pixel in the image memorizes the number of the Sub-Texture, to which it belongs.

3.1.3.12 Procedure LB: Local Brightness of the Background

Two types of the local brightness functions $Bb_i(z)$ are used. For the first type (zero order)

$Bb_i(z)$ is identically equal to the input brightness value Bb_i at the point Z_i .

For the second type (first order) $Bb_i(z)$ is equal to $L_i(z)$, where $L_i(z)$ is the linear function, such that

$$L_i(Z_i) = Bb_i$$

and Li provides the best approximation of the input brightness values at the N nearest to Z_i ACs. The choice of the type of the local brightness function is determined by the flag LBF: LBF is zero for the zero order and LBF is one for the first order of the functions $Bb_i(z)$. Here N is an integer valued tuning parameter.

Typical value of N is 4 or 9: usually the ACs form a regular or an almost regular grid, and the nearest neighbors are taken at each point Zi to construct the linear function Li(z).

3.1.3.13 Procedure WB: Weights for the Background

As implied by the form of the expression, the weights $WR(d(z, Z_i))$ depend only on the distance $d(z, Z_i)$ from the point z to the AC Zi. The model function of one variable WR is specified by three tuning parameters UDB and UB2, UDB > UB2 > 0, and BVS (Background Weight smoothness), 0 < BVS < 1, and is defined as follows:

$WR(t) = 0$ for $|t| > UB1,$

$WR(t) = 1$ for $|t| < UB2,$ and

$WR(t) = BVS(3v^2 - 2v^3) + (1 - BVS)v,$ for $UB2 < |t| < UB1,$

where $v = (|t| - UB2)/(UB1 - UB2).$

See the Figure below.

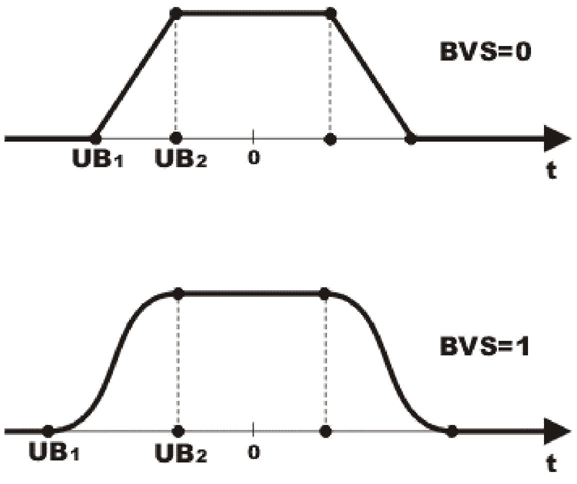


Figure 18 — Synthesized Texture Background weight functions

3.1.3.14 Procedure SE: Signal Expansion

Let D denote the domain of the Synthesized image, with “cuts” along all the separating Lines PL_i. For any two points z_1, z_2 in D the distance $dd(z_1, z_2)$ is defined as the (Euclidean) length of the shortest path, joining z_1 and z_2 in D and avoiding all the cuts PL_i. See the figure below.

Note: It might be assumed that the influence of the color at z_1 to the color at z_2 decreases as the distance $dd(z_1, z_2)$ increases. However, a precise computation of the distance dd is a rather complicated geometric problem. Consequently, we use instead of the distance $dd(z_1, z_2)$ its approximation $d(z_1, z_2)$, which is computed through a “signal expansion algorithm”, as described below.

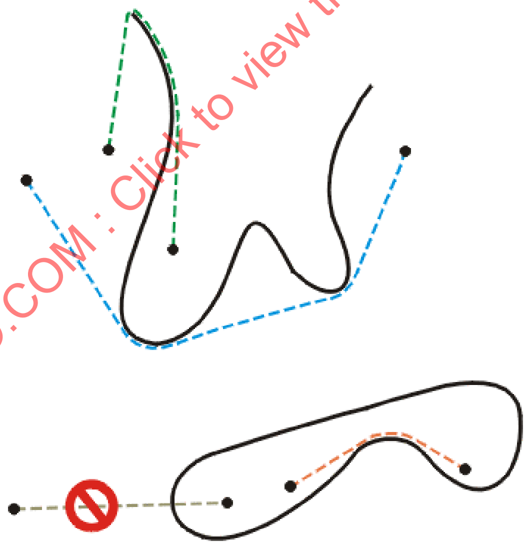


Figure 19 — Shortest path between points

The block SE computes the distance $d(z_1, z_2)$ for any two points z_1 and z_2 in the image plane. The algorithm is not symmetric with respect to z_1 and z_2: in fact, for a fixed point z_1, the distance $d(z_1, z_2)$ is first computed for any pixel z_2 of the image. Then an additional routine computes $d(z_1, z_2)$ for any given z2 (and not necessarily a pixel).

Below the notion of a “neighboring pixel” is used. It is defined as follows: for z not a pixel, the four pixels at the corners of the pixel grid cell, containing z, are the neighbors of z. For z a pixel, its neighbors are all the pixels, whose coordinates in the pixel grid differ by at most one from the coordinates of z.

Below we assume that a certain data structure is organized, in which to any pixel p on the image plane a substructure is associated, allowing to mark this pixel with certain flags and to store some information, concerning this pixel, obtained in the process of computation. We do not specify here this data structure, using instead expressions like "the pixel p is marked", "the pixel p memorizes…" etc.

Now for z_1 and z_2 given, the distance $d(z_1, z_2)$ is computed in the following steps:

For any pixel p the distance u(p) to the *separating* Line PL_i is computed and stored at this pixel. The computation of u(p) is performed by the procedure **DL**, described above, applied only to separating poly-links PL_i.

Those pixels p, for which u(p) < FU, are marked as "forbidden" pixels. The forbidden pixels are excluded from all the rest of computations, and those pixels that are not forbidden, are called below "free" ones. Here FU is a tuning parameter.

Now the proper "signal expansion" starts. In the first step each of the free neighbor pixels of z_1 is marked, and this pixel memorizes its Euclidean distance from z_1 as the auxiliary distance dd, to be computed. Generally, in the k-th step, any free unmarked pixel p, at least one of whose free neighboring pixels was marked in the previous steps, is marked. This pixel memorizes as its auxiliary distance dd(p) from z_1, the minimum of dd at the neighboring free pixels plus the Euclidean distance of p to the neighboring pixel, at which the minimum is attained. This process is continued the number of steps, equal to the maximal dimension of the image (in pixels). After it is completed, each free pixel p on the image plane memorizes its auxiliary distance dd(p) from z_1.

Now for any given point z_2 on the image plane, its distance $d(z_1, z_2)$ from z_1 is computed as maximum of D1 and D2, where D1 is the Euclidean distance of z_2 to z_1 , and D2 is the minimum over the free neighboring to z_2 pixels p, of dd(p) + the Euclidean distance of z_2 to p.

This completes the computation of the distance $d(z_1, z_2)$.

See the Figure below.

Note: The tuning parameter FU determines the size of a neighborhood of the separating Line, where all the pixels are marked as forbidden. Taking any value of FU, larger than 0.8, excludes a possibility of signal expansion crossing separating lines. Indeed, for any two neighboring pixels, which are on different sides of a separating line, at least one is closer to the line than 0.8 and hence is marked as forbidden. To provide stability of finite accuracy computations a bigger value of U may be taken. However, in this case signal expansion will not pass a "bottle-neck" between two separating lines, which are closer to one another than 2FU. Normally such regions will be covered by the cross-sections of these lines. However, a sub-pixel grid can be used to guarantee that signal expansion passes thin "bottle-necks"

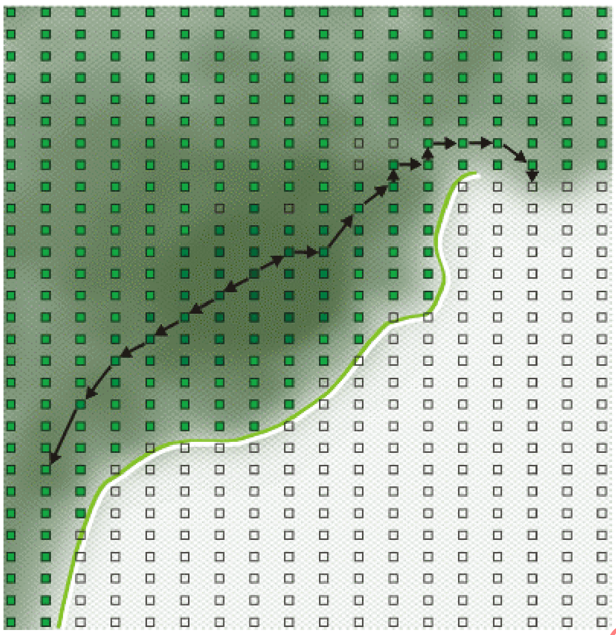


Figure 20 — Signal Expansion in Synthesized Texture rendering

Various implementation issues:

Computation of the distance d(z_1, z_2) and its usage inside the Background grid interpolation form one of the central parts of the Synthesized image reconstruction. Consequently, the efficiency of the implementation of these blocks is crucial for an overall efficiency of the reconstruction.

A straightforward implementation of the signal expansion algorithm, as described above, is not optimal. However, a number of rather natural and simple modifications bring the efficiency of the signal expansion to only few operations per pixel.

3. Multi-scale implementation. The image is subdivided into blocks in 2 – 3 scales (for example, blocks of 16x16, 8x8 and 4x4 pixels). First signal expansion is performed between the highest scale blocks (say, 16x16), exactly as described above. Forbidden are the blocks, crossed by separating Lines. In the second stage the forbidden blocks are subdivided into 8x8 sub-blocks, and the expansion is performed for them. The new forbidden sub-blocks are subdivided into the 4x4 ones, and the expansion is repeated. In the last stage the expansion is completed on the pixels level.

j) For an application to the Background grid interpolation, the distance d(z1, z2) has to be computed only till it riches the threshold UDB (since for larger distances the weight functions vanish). This fact allows one to restrict the number of steps in signal expansion to UDB + 1.

k) Signal expansion and memorization of the distances at the free pixels can be implemented for all the ACs at once (especially since the above distance restriction usually makes for any pixel only the information relevant, concerning a few neighboring Background grid points).

l) In the process of signal expansion, all the mathematical data required in the interpolation block (like Euclidean distances and weight functions) can be computed incrementally in a very efficient way.

3.1.3.15 Procedure MRP: Margin Representing Points

This Procedure constructs a grid of representing points on the margins of all the Lines together with the Background brightness values (APs) at these points. Later the constructed margin points are used (together with the original ACs) in the interpolation process in the block BB.

The margin representing points Mz_j are produced in the following steps:

4. On each Line, the points wk are built with the distance UM1 from one another, starting with one of the ends (the distance is measured along the Line). The last constructed point on each Line may be closer to the end Terminal Point of this Line, than UM1.

5. At each wk the line lk orthogonal to the Line and intersecting it at wk is drown. If wk turns out to be a vertex of the Line with a nonzero angle between the adjacent Line Segments LS (links), or a crossing, lk is taken to be the bissectrice of the corresponding angle.

6. On each line lk two points (one point in the case of the bissectrice of the crossing joint angle) are chosen at the distance $UM2 \cdot W(w_k)$ from the intersection point wk of lk with the Line (from the crossing wk, respectively). All the chosen points, in a certain chosen order, form the output margin representing points Mzj.

7. At each margin representing point Mzj constructed, as described above, the corresponding margin Background brightness value Bbj is computed by

$$Bb_j = tA + (1-t)B$$

where A and B are the margin values (colorFarLft or colorFarRgt, respectively) of the cross-sections at the ends of the Line Segment $S(Mz_j)$, nearest to the point Mz_j, and $t = t(Mz_j)$.

$S(Mz_j)$ and $t(Mz_j)$ are computed by the Procedure **DL.**

In the current Procedure UM1 and UM2 are tuning parameters (the first one absolute and the second relative to the width), satisfying UM1 < UB1, 1 < UM2 < UL1.

See the figure below.

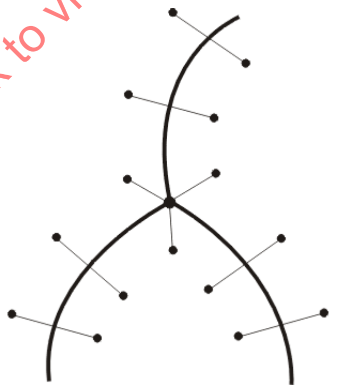


Figure 21 — Margin representing points

The four figures below illustrate influence of different elements of the Background for different values of tuning parameters.

Figure 22 — shows an extrapolation of the margin values of the Line.

Figure 23 — illustrates the influence of the Area Color Points (ACs).

Figure 24 — shows representing points with a larger parameter UB1.

Figure 25 — shows a Patch influence.

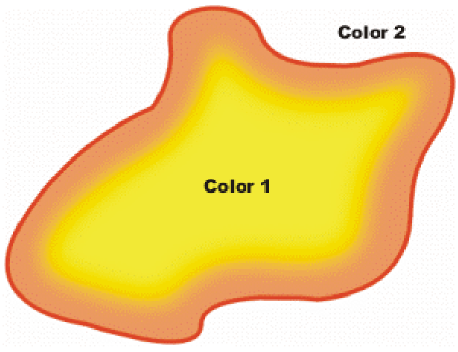


Figure 22 — Line influence

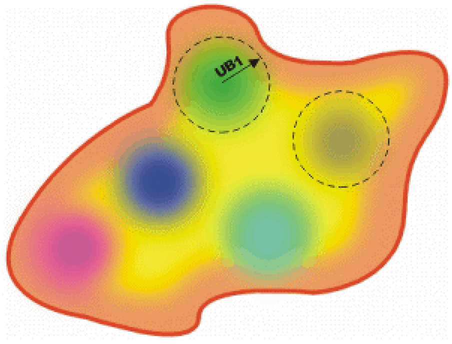


Figure 23 — Area Color influence

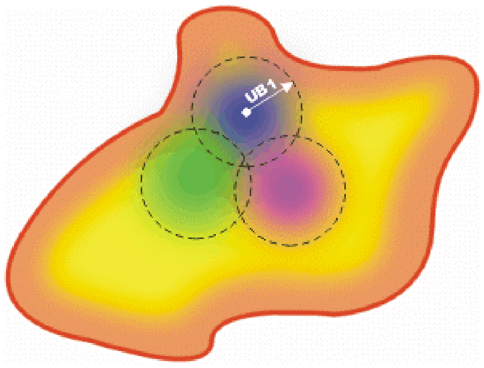


Figure 24 — Effect of larger UB parameter

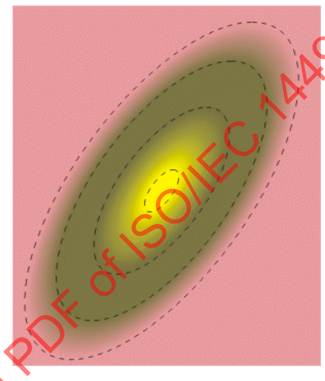


Figure 25 — Effect of Patch

3.1.3.16 Tuning Parameters

All the tuning parameters, listed below, are not changed in the process of sending and playing of images and animations. However, their tuning for specific visual displays (and for specific classes of images) may improve the visual quality and the player efficiency.

UL1 – the distance from the Lines (as a proportion to the width of the line), within which the CP (cross-section) brightness BL(z) is computed. It coincides with the exterior size of the lines averaging region. Used in the Proceduress BL and WL.

UL2 – the interior size of the lines averaging region (as a proportion to the width of the line). Used in the Procedure WL

UP1 and UP2 are the corresponding parameters for Patches (referring to the relative distance with respect to the Patch size). Used in the Procedures BP and WP.

UB1 and UB2 are the (absolute) size parameters for the weight functions of the Area Color Points (ACs). The parameter BVS characterize the smoothness of these weight functions. Used in the Procedure WB.

The flag LBF specifies the order (0 or 1) of the local brightness representation. Used in the Procedure LB.

The integer parameter N is the number of neighboring ACs taken to define the local linear approximations of the brightness. Used in the Procedure LB.

FU is the distance of the separating lines at which pixels are marked as “forbidden”.

UM1 and UM2 are the absolute and the relative to the width parameters in construction of margin representing point. Used in the Procedure MRP.

D > 1 is the parameter, defining the “asymmetry” of the distance to the Line Segment, computed in its side areas. Used in the Procedure DDL.

DE > 0 is a parameter, defining the length of the end area of Lines. Used in the Procedure BL.

3.1.4 The Skeleton

3.1.4.1 General

The Skeleton is a set of lines used for transferring motion to the elements of the texture during animation, through the process of Texture Warping (§3.1.5).

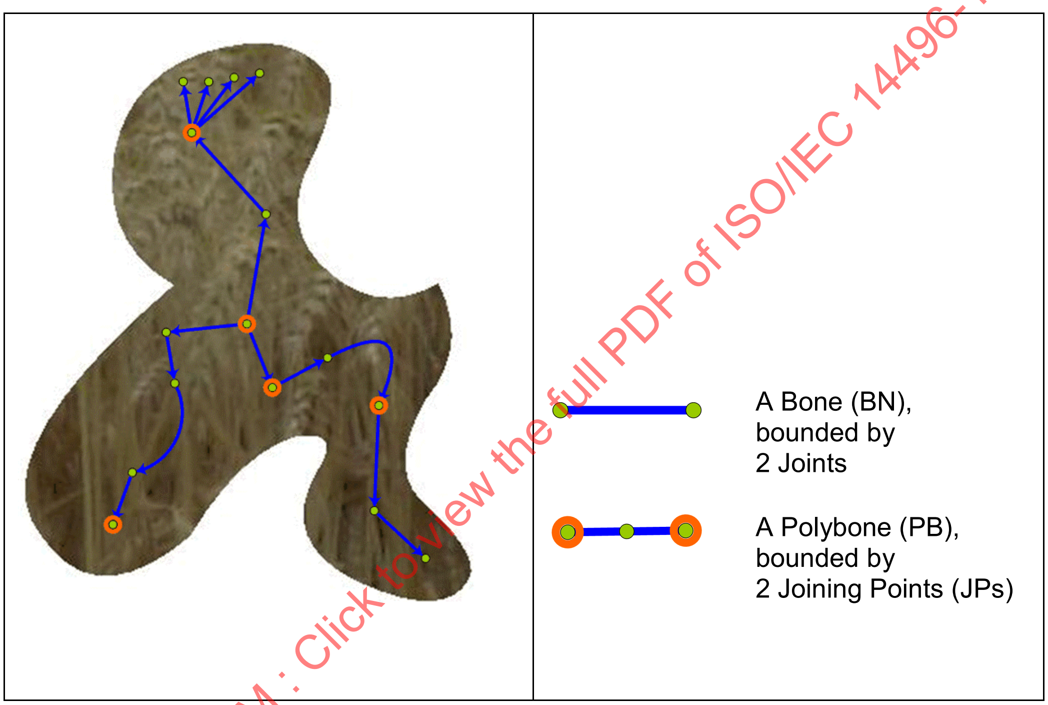

A Bone (BN), bounded by 2 Joints
A Polybone (PB), bounded by 2 Joining Points (JPs)

Figure 26 — Synthesized Texture Skeleton Topology and Geometry

3.1.4.1.1 Topology

The skeleton **topology** is based on a finite directed graph[3], which includes a set of points called **Joints** (graph vertices), and a set of ordered pairs called **Bones** (BNs), which are graph "links" that connect specific pairs of joints directionally.

Series of consecutive connected bones are grouped in **Polybones** (PBs). Polybones terminate in **Joining Points** (JPs).

The Skeleton is morphologically similar to the Texture, where Bones (BNs) correspond to Line Segments (LSs), Polybones (PBs) correspond to Lines (LNs), and Joining Points (JPs) correspond to Terminal Points (TPs).

3) A **finite directed graph** is a finite set *V* of nodes called **vertices**, and a set of ordered pairs (v_{start}, v_{end}) called **links** that connect some pairs of vertices where v_{start}, and v_{end} are in *V*. A graph describes the connectivity pattern of its vertices, and has no inherent geometry.

However note the direction of polybones is significant to their function, while the direction of texture Lines is arbitrary. Additionally, PB need not necessarily terminate in a JP, while all LNs must terminate in TPs.

Some of the skeleton's Joining Points may be attached to a particular Polybone by a **Bond**. A bond forces the initial relative position between a particular JP and a particular PB to be maintained throughout animation.

3.1.4.1.2 Geometry

During authoring and animation, the skeleton is initially laid out on a the plane of the texture which it is associated with. When the skeleton's pose changes, i.e. when it moves relative to itself, the skeleton's components change their location and shape, but they always remain on the texture plane.

In addition to skeleton topology, Skeleton data contains one specific geometry, used later:

- as the default mapping between the layers in the object's texture and the polybones in the skeleton topology.
- as the skeleton's initial pose during authoring and/or playback.

3.1.4.1.3 Association with Layers

During authoring and encoding, every polybone is associated with exactly one texture Layer. During decoding, this allows associating in bulk each texture primitive in that layer, with a single polybone.

3.1.4.2 Primitives and Properties

The following primitive classes are used to store skeleton data, and are the targets of the Skeleton Decoding process described in §4.7.

3.1.4.3 Skeleton

3.1.4.3.1 Syntax

```
class Skeleton
{
  char name[16]
  int  orientation;
  int  width;
  int  height;

  aJP[] aJP;      // Joining Points
  aPB[] aPB;      // Poly-Bones
  int   nJP = 0;
  int   nPB = 0;
 }
```

3.1.4.3.2 Semantics

Name	Description
name	The skeleton's name.
orientation	A bogus attribute of the skeleton topology, allowing a single topology to have several distinct instances, each with a different "orientation". This attribute can then be used in matching or adapting an adequate animation for a specific "orientation". This is currently used when authors choose to "flip" (mirror) a skeleton - the "orientation" of the skeleton is then changed from 0 to 1. Animations created for this skeleton are then adequate only for topologies whose "orientation" is 1. alternatively, animations created for orientation=0 may to be "flipped" in order to be made adequate for the skeleton whose "orientation" is 1.

Name	Description
width	The width of the skeleton's plane in pixels.
height	The height of the skeleton's plane in pixels.
aJP	Array of Joining Point data for this skeleton.
aPB	Array of Polybone data for this skeleton.
nJP	Number of elements in aJP.
nPB	Number of elements in aPB.

3.1.4.4 Joining Point (JP)

3.1.4.4.1 Syntax

```
class JP                    // Joining Point
{
  int iJP;

  float x, y;

  int iJpBond = -1;
  int iBrBond;

  BR   aBR[];               // branches from this JP
  int  nBR = 0;             // number of branches from this JP
}
```

```
class BR
{
  int   iPB;              // index of PB that branches from this BR
  bool  bOut;             // is this branch an "out-branch"?
}
```

3.1.4.4.2 Semantics

Name	Description
iJP	The index of this JP in the skeleton's aJP array.
x, y	The coordinates of this JP in the skeleton's plane.
iJpBond, iBrBond	let "BondPB" be the polybone that this JP is (optionally) bonded to. Then: iJpBond is the index of the starting JP of BondPB. If iJpBond =-1 then this JP does not have a bond. iBrBond is the index of the BR that BondPB is branching-out from.
aBR	The array of branches from this JP.
nBR	The number of branches in aBR.
class BR:	
iPB	The index of the Polybone that branches from this BR.
bOut	Is this branch an "out-branch"?

3.1.4.5 Polybone (PB)

3.1.4.5.1 Syntax

```
class PB                             //  Polybone
{
  int iPB;
  int iBegJP, iBegBR;
  int iEndJP, iEndBR;
  int iLY;
  int influence;

  BN  aBN[];
  int nBN = 0;
}
```

3.1.4.5.2 Semantics

Name	Description
iBegJP, iEndJP	The indexes of the JPs in which this Polybone begins and ends.
iBegBR	The index of the branch in aJP[iBegJP] from which this Polybone begins.
iLY	The index of the Layer that this polybone is associated with.
influence	The range of this polybone's influence on layer iLY. 0=infinite influence; else=influence range from polybone, given in pixels.
aBN	The array of Bones comprising the geometry of this Polybone.
nBN	The number of elements in aBN.

3.1.4.6 Bone (BN)

3.1.4.6.1 Syntax

```
class BN                    // Bone
{
   float vX, vY;
   float hgt;
}
```

3.1.4.6.2 Semantics

Name	Description
vX, vY	The x,y offset of this Bone's end point from its starting point, in pixel units.
hgt	The height of the curve of this BN.

3.1.5 Texture Warping

3.1.5.1 General

During animation, the Texture is "warped" by re-calculating the location of the texture primitives. This calculation is based on the current geometry of the skeleton.

The geometry of the skeleton changes based on 3 motion patterns of bones in the skeleton:

- **Rotation** around one of the bone's end points (joints).
- **Stretching** (or shrinking) of a bone along the cord connecting its joints.
- **Bending** a bone, by modifying the height of its arc.

The animation mechanism is comprised of:

- An **influence scheme**, specifying the layers influenced by each bone of the skeleton.
- An **influence zone**, specifying the points in the layer's texture that are moved.
- A **motion transfer model**, specifying the way in which the skeleton motion influences the elements of the layer's texture.

Figure 27 bellow demonstrates the effect of bone movement on the texture.

A photo of a bottle with a background was converted to Texture. After conversion, a single Layer (the bottle) was cut-out of the image, and a single-layered object was defined. Then, a 3-bone polybone was attached to the object.

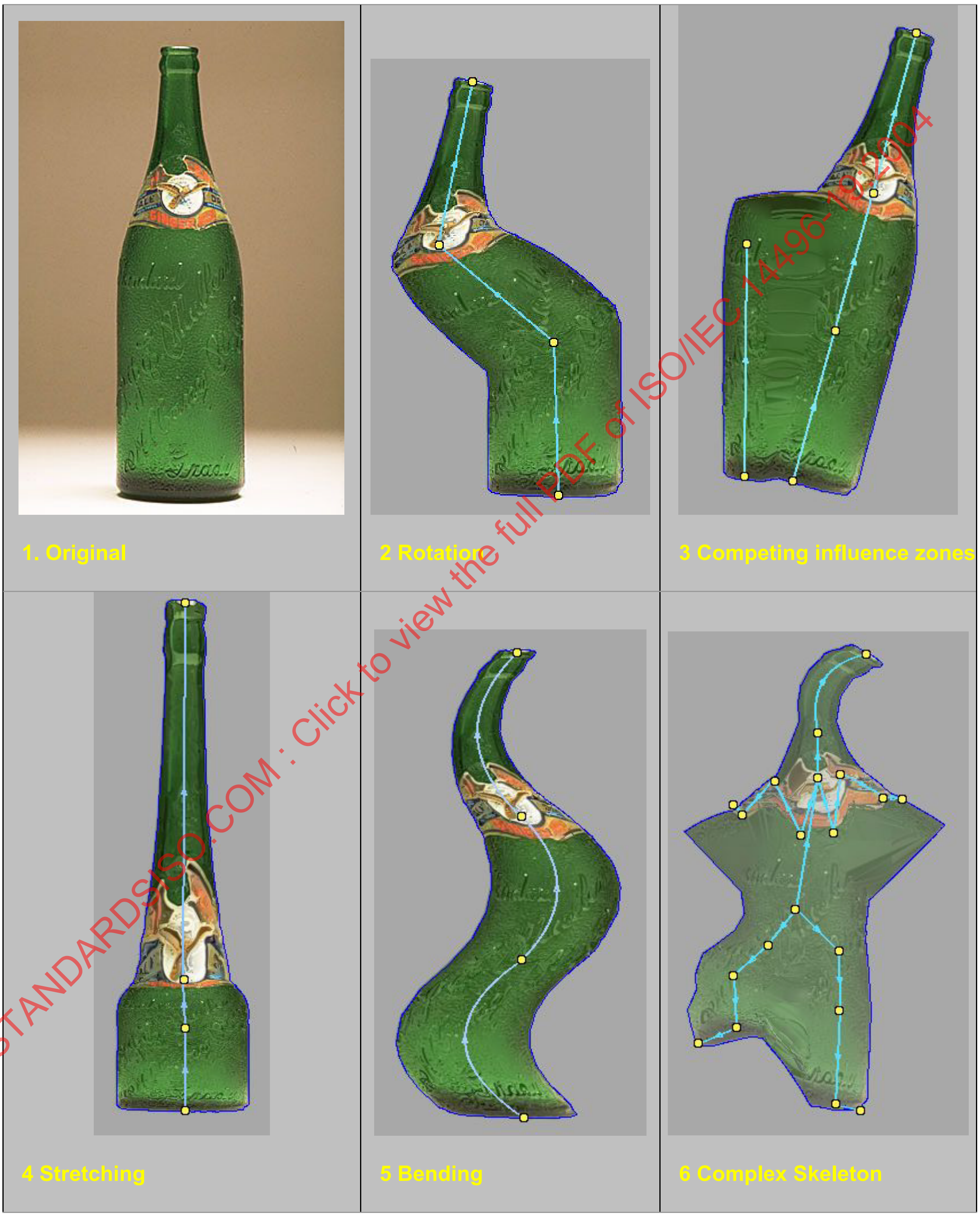


Figure 27 — The effect of bone movement on the texture.

1. The source image before conversion to Synthesized Texture
2. Rotation of each bone separately (with no stretching or bending)
3. Rotation of the polybone as a whole, with another bone added. Note the texture in the added bone's influence zone remains static, causing a stretching effect between it and the central polybone
4. Stretching of the top bone and shrinking of the two bottom bones (with no rotation or bending)
5. Bending of each bone separately (with no rotation or stretching)
6. A complex skeleton (resembling humanoid) is attached to the object, and rotation, stretching/shrinking, and bending are all applied

3.1.5.2 The Influence Scheme

The basic influence scheme of the skeleton is a list, specifying the layers affected by each bone in the skeleton.

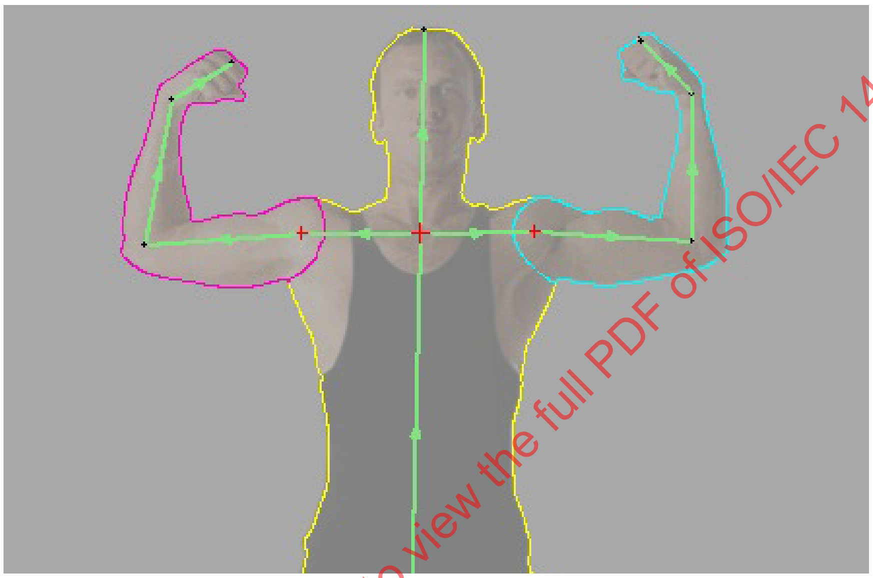


Figure 28 — The associatioin of layers with polybones.

The object includes 3 layers, dividing the object's skeleton into 3 polybones, whose default influence zones are their respective layers themselves.

Additionally, "artificial" influence regions can be imposed for a polybone, consisting of all the points in the layer that are closer to this polybone than a certain threshold. This threshold is chosen according to the actual shape of the object to be animated.

The bone association scheme determines each Texture primitive's **controlling bones**. Only these bones affect the primitive during motion transfer.

The set of bones controlling a primitive is constructed according to the following scheme:

A. The primitive's controlling polybones are determined:

- Each primitive is normally a member in a specific layer in the texture.
- Each layer is normally controlled by one or more polybones.
- When the primitive's layer is associated with multiple polybones, the "influence-range" values of the polybones in question are used to weight the relative influence of each polybone on the primitive during motion transfer.

Otherwise, if the primitive's layer is associated with a single polybone, this is its controlling polybone.

B. The primitive is associated with a specific controlling bone within each controlling polybone:

- The bone closest to the primitive's center is found for each controlling polybone, and added to the primitives set of controlling bones.

C. Too-far bones are excluded from influence:

When the primitive has multiple controlling polybones, bones found above that are farther then the respective polybone's influence-range are excluded from the set.

3.1.5.3 The Motion Transfer Model

The animation of texture is based on translating the motion of the bones of the skeleton into the motion of the nearby **texture points**:

- The Line Points (LP), defining the course of Lines.
- The Area Color Points (AC), defining coloring between Lines.
- The centers of Patches (PA).

Since the geometry of the entire Texture is completely defined by the position of these points, the skeleton's state completely defines the motion of the influenced layers along time.

The following general scheme is used to translate the skeleton motion to the nearby points:

- A coordinate frame (§3.1.5.4) is constructed, comprising a coordinate systems around each bone in the skeleton.
- An influence region (§3.1.5.2) for each bone in the skeleton is defined.
- These coordinate systems and influence regions follow the motion of the skeleton.

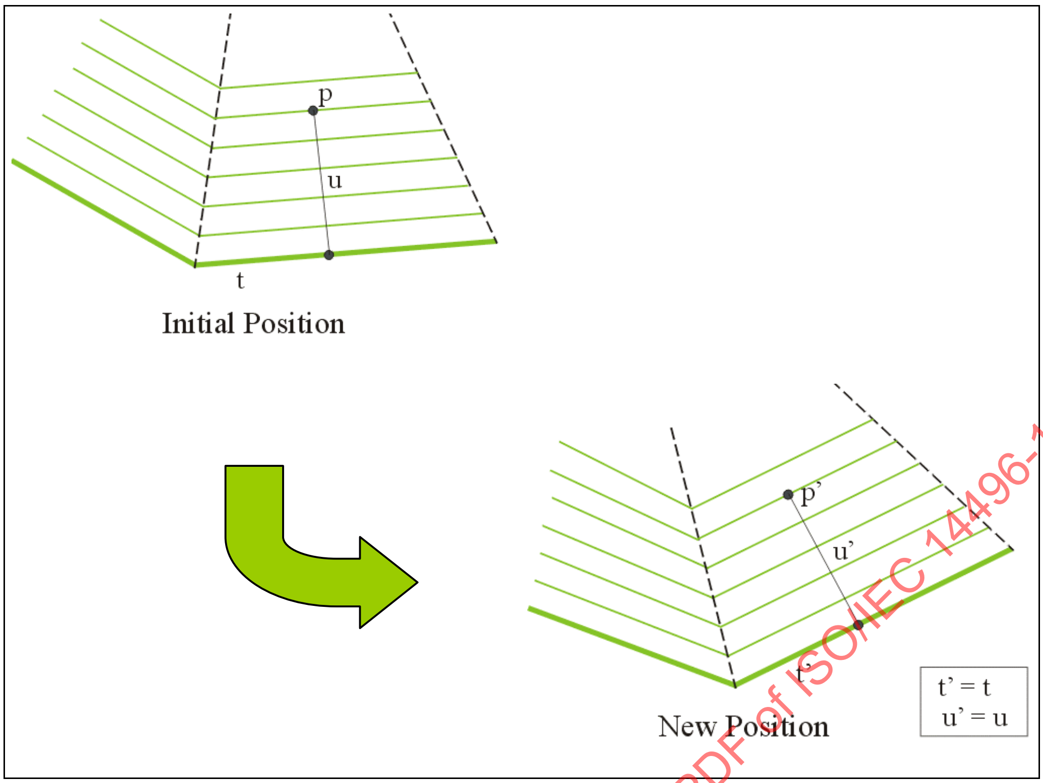


Figure 29 — The effect of skeleton motion on a texture point: the polybone affecting p is displaced (right and downwards), and its geometry changed (the right bone has rotated causing the angle with the left bone to become smaller).

Then, in order to define the position of a certain texture point *p* corresponding to a given motion of the skeleton, the following steps are performed:

- If *p* does not belong to any of the influenced layers, it does not move.
- If *p* belongs to a certain influenced layer, its coordinates with respect to the corresponding bone are computed once.
- During animation, whenever the skeleton moves, a new point *p'* is calculated, whose coordinates with respect to the current state of the bone are the same as the coordinates of *p* with respect to the original state of the bone.

3.1.5.4 The Coordinate Frame

3.1.5.4.1 Bone Coordinate System

The coordinate frame of the Skeleton consists of special coordinate systems, associated with each of the bones in the skeleton, and of the **influence regions** of these bones.

A coordinate system *(U, T)* is defined for each bone in the skeleton, so that for any texture point *p*:

- *u* is the distance of *p* from its **nearest bone** *b*.
- *t* is the coordinate of the projection of *p* onto *b*, i.e. the distance along *b* of the projection of *p* from one of the end points of *b*. The projection is not exactly the orthogonal one; instead it takes into account the bisectrial lines of the angles between the adjacent bones in the polybone.

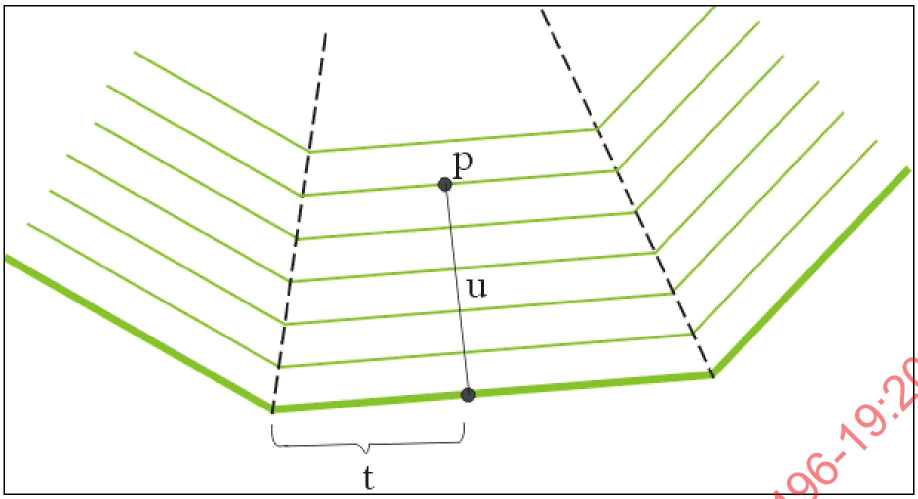


Figure 30 — A polybone's coordinate scheme, and point *p*'s coordinates *(u, t)*.

p's nearest bone *b*, as well its coordinates *(u, t)* relative to *b*, are pre-computed.

The new position of *p* is calculated according to the rule that its coordinates *(u', t')* relative to the moved *b* remain the same as the initial coordinates *(u, t)*.

A bone's coordinate systems *(U, T)* as defined above adheres to the bone regardless of the bone's movement relative to the skeleton and to the SynthesizedTexture's world. Thus, the new coordinates *(u', t')* for a texture point *q* are defined by the same expressions as above:

- *u'* is the distance of *q* from its transformed nearest bone.
- *t'* is the coordinate of the projection of *q* onto the transformed bone, i.e. the distance, along that bone, of the projection of *q* from one of the end points of the bone.

3.1.5.4.2 Non-Unique Projections

If a bone in the skeleton has a complicated geometric shape, and if an associated texture point *p* is relatively far from this bone, the projection of *p* onto the bone, and hence the distance *u*, the coordinate *t* and the rotation angle *w*, are not uniquely defined. This, in turn, leads to numerical instability of the relevant computations.

This problem is overcome since:

- Only the points within a bone's influence region (§3.1.5.2) are actually displaced. In implementations these regions are taken small enough to provide uniqueness in the above algorithm.
- The skeleton's kinematical model restricts motion so that complicated or unstable shapes cannot be produced.

3.1.5.5 Bending and Stretching of Bones

3.1.5.5.1 General

In the general case, the bones of the Skeleton are not only displaced in the animation process, but may also be stretched and bent[4]. This is made possible by using parabolic segments as bones.

Thus, in addition to its position determined by its two end points, each bone in the skeleton may have the following three additional attributes:

- Bone Stretching (*a*).
- Bone Bending amplitude (*b*).
- Bone Bending direction (*f*).

The effect of stretching and bending a bone on the animation of its associated texture points is linear and thus natural.

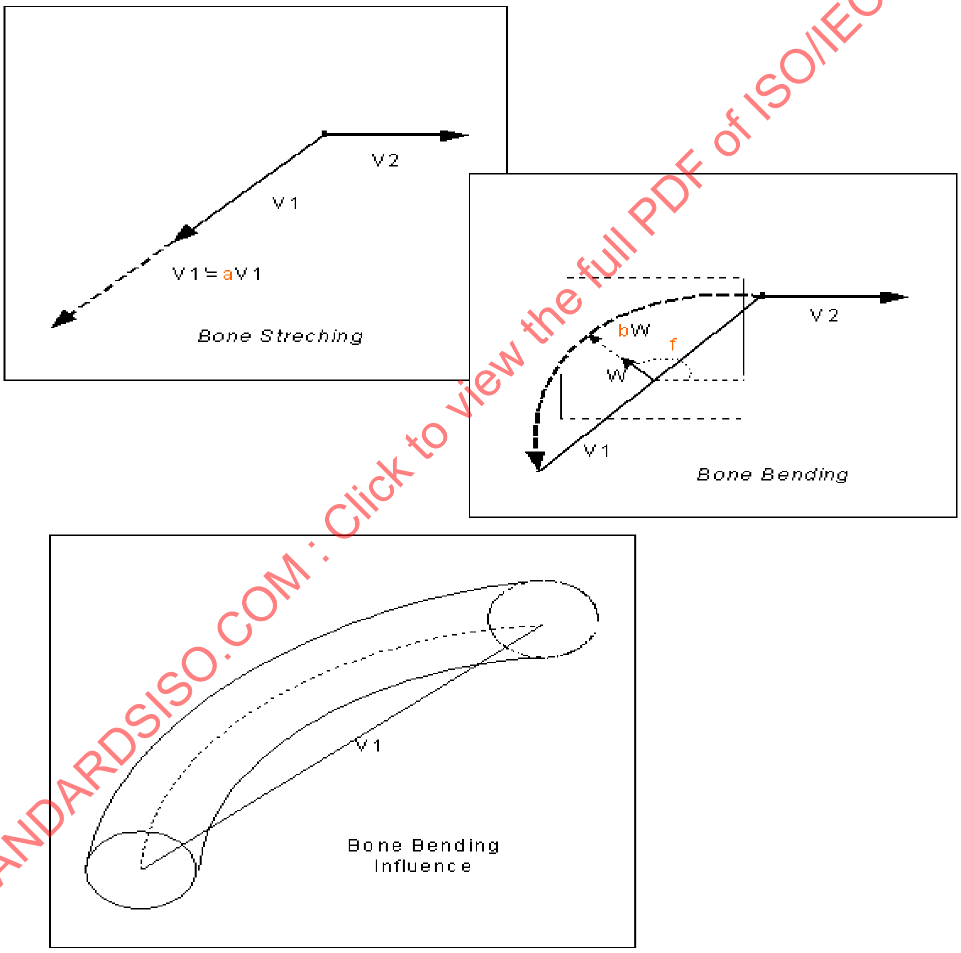


Figure 31 — The effect of bending and stretching on the texture.

4) In a realistic animation of a human body, stretching and bending of bones is usually unnecessary, since real human bones do not normally stretch or bend. However, in non-human or less realistic animations these motion patterns are common. Moreover, stretching and bending allow artists to compensate for possible distortions produced by the pose of the animated object in the original image. Additionally, these animation patterns allow for various important visual effects by means of this essentially 2D mechanism, including 3D-like motion effects.

Stretching (or shrinking) a bone by a factor *a* stretches its coordinate system *(U, a*T)*, thus causing the position of associated texture points to be proportionally displaced, causing a stretching effect to the entire associated texture.

Bending a bone bends the bone's coordinate system, but since each texture point's *(t, u)* coordinates remains constant, a bending effect of the entire associated texture is achieved.

3.1.5.5.2 The Effect on the Texture

In the figure above, an orthonormal coordinate system, related to the initial joint of the bone, is used. V1 denotes the bone vector, V is a vector of a generic given point, while V' denotes the vector of this point after transformation.

Stretching is defined by the stretching parameter a. For the bone vector, V1' = a*V1. For any vector V: represent it in the coordinate system of the bone, as V = pV1 + qV2, where V1 is the bone vector, and V2 is the unit vector, orthogonal to the bone. Then the new vector V' is given by V' = a*pV1 + qV2.

In other words, a times linear stretching is performed on all the space in the direction of the bone vector V1. Bending is defined by the bending amplitude parameter b and the bending direction parameter f. The bending direction is given by a unit vector W at the central point of the bone, which is orthogonal to the bone vector V1, and it is completely determined by the direction f.

To compute the result of a bending on any vector V, representation of the vector V in the coordinate system as above is used:

If V = pV1 + qV2 then V' = pV1 + qV2 + b*p(1-p)W, for p between 0 and 1, V' = V, for p outside of the interval [0,1], i.e. for the projection of the point V on the bone's line outside the bone.

3.1.6 Object Animation

3.1.6.1 General

The animation effect of Synthesized Texture Objects is achieved by changes made to the appearance of the object's Texture along time.

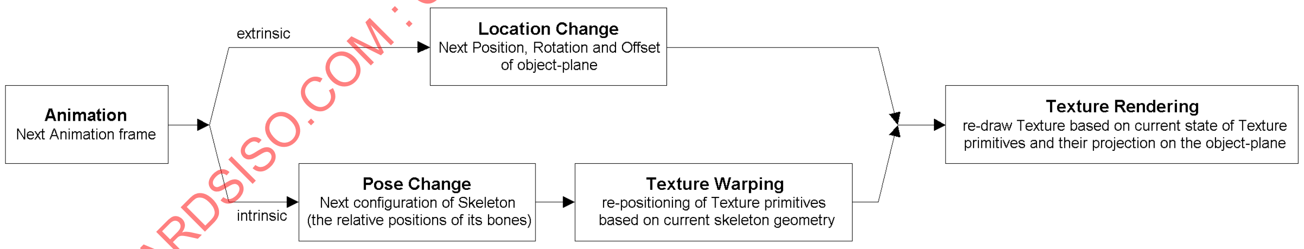

extrinsic
Location Change
Next Position, Rotation and Offset of object-plane
Animation
Next Animation frame
Texture Rendering
re-draw Texture based on current state of Texture primitives and their projection on the object-plane
Pose Change
Next configuration of Skeleton (the relative positions of its bones)
intrinsic
Texture Warping
re-positioning of Texture primitives based on current skeleton geometry

Figure 32 — The animation mechanism

The animation mechanism uses a sequence of frames describing the object's state along a timeline. Each frame describes the Object's **location** and **pose** at a corresponding point in time.

Changes in the location include changes to the position and rotation of the object-plane and the offset of the Object (i.e. its Skeleton) on that plane.

Changes in the pose include changes in the relative positions of the bones in the Skeleton, which is laid out on the object-plane. These induce changes in the position of the Texture primitives on the object-plane through the process of Texture Warping.

Finally, the Texture is re-rendered on the visual plain based on the absolute location of the Texture primitives in the world, and its projection on the visual plane. For Scenes which Include a Camera Scenario, the position and viewing angle of the Camera is additionally considered.

3.1.6.2 Object State Information

Each frame in the Synthesized Texture **Animation** describes the object's geometrical state:

1. Location – provides the **extrinsic** motion of the object:
   - the current **position** and **rotation** of the object's 2D plane in the SynthesizedTexture's 3D world.
   - the current **offset** of the object on its 2D object-plane.
2. Pose – provides the **intrinsic** motion of the object:
   - the current geometry of the object's **Skeleton**, as it is laid-out on the 2D object plane.

3.1.6.3 Frame Sequence Information

An Animation is a sequence of **Frames** describing the changes in the object's state along a timeline.

The frame sequence of an object's animation is described and encoded using a subset of the frames in the animation. Frames in this subset are called **KeyFrames**.

The frame sequence of an object is divided into one or more sub-sequences called **Runs**. A run is bound by at least a start and end keyframe. The end keyframe of a run is called a **run-end** keyframe. All other keyframes in a run are called **run-through** keyframes,

All frames in a run that are not keyframes are called **in-between** frames. The geometrical properties of in-between frames are not explicitly described in the bitstream, and are derived instead, in the process called **tweening** (from "between"). In tweening, linear interpolation is performed on the geometrical properties of two adjacent keyframes, yielding the values of these properties for the in-between frames.

Frames between runs are called **static** frames. In static frames, the object remains absolutely static, preserving its state at the end of the preceding run.

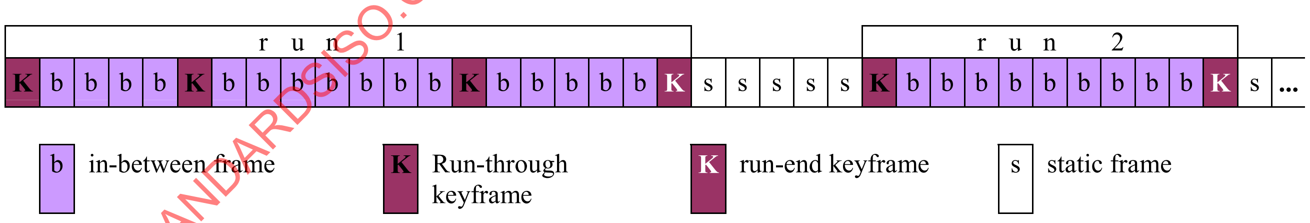


Figure 33 — Synthesized Texture Animation: Runs, Keyframes and inbetweens

3.1.6.4 Primitives and Properties

The following primitive classes are used to store animation data, and are the targets of the Animation Decoding process described in §4.8.

3.1.6.5 Animation

3.1.6.5.1 Syntax

```
class Animation
{
  KF[] aKF;
  int  nKF;
}
```

3.1.6.5.2 Semantics

Name	Description
aKF	The KFs in this Object Animation.
nKF	The number of KFs in aKF.

3.1.6.6 KeyFrame (KF)

3.1.6.6.1 Syntax

```
class KF          // Object Animation Key Frame
{
  int    iFrame;
  bool   bRunEnd;

  bool   bHasState3D;
  float  posX;
  float  posY;
  float  posZ;
  float  rotX;
  float  rotY;
  float  rotZ;

  float  locX;
  float  locY;

  Skelton skl;

  // auxiliary
  int iFrameDiff;
}
```

3.1.6.6.1.1 Semantics

Name	Description
IFrame	The index of the animation frame that this keyframe describes.
BRunEnd	Is this a run-end KF? Otherwise this is a run-through KF. Run-end KFs are coded relative to the object's base KF, while run-through KFs are coded relative to their predecessor KF. This is so because the contents of a run-through KF are by definition related to the contents of its predecessor KF, while the contents of a Run-end KF are not.
posX, posY, posZ	The position of the origin of the object's 2D plane in the SynthesizedTexture's 3D world, in worldUnits.
rotX, rotY, rotZ	The rotation of the object plane about each of the 3D world's axes, in radians.
locX, locY	The offset of the object (i.e. of its bounding rectangle's top left corner) on the object plane, in pixels.
Skl	The skeleton whose geometry is applied to this keyframe. Joint locations are encoded based on the changes in the angles between the bones of the skeleton between this keyframe and the previous one.
IFrameDiff	The difference between this KF's frame number and that of the previous KF.

3.1.7 Camera Scenario

3.1.7.1 General

The Camera Scenario describes the behavior of the SynthesizedTexture Camera along time. Similar to Object Animation, The time-based scenario of the Camera is based on a sequence of **Frames**.

Each frame describes the camera's 3D state:

- **Position** - the position (translation) of the origin of the camera's plane in the SynthesizedTexture's 3D world.

- **Rotation** - The rotation of the camera's plane about the 3D world's coordinate system.

Identical to Object Animation (§3.1.6), the Camera Scenario is based on a keyframe mechanism, which uses a sequence of **KeyFrames** to describe the Camera's state throughout all the frames in the scenario.

3.1.7.2 Primitive and Properties

The following primitive classes are used to store animation data, and are the targets of the Camera Decoding process described in §4.9.

3.1.7.3 Camera

3.1.7.3.1 Syntax

```
class Camera
{
  KF[] aKF;
  int  nKF;
}
```

3.1.7.3.2 Semantics

Name	**Description**
AKF	The KFs in this Camera Scenario
NKF	The number of KFs in aKF.

3.1.7.4 KeyFrame (KF)

3.1.7.4.1 Syntax

```
class KF         // Camera Key Frame
{
  int     iFrame;

  float   posX;
  float   posY;
  float   posZ;
  float   rotX;
  float   rotY;
  float   rotZ;

  // auxiliary
  int iFrameDiff;
}
```

3.1.7.4.2 Semantics

Name	Description
lFrame	The index of the camera frame that this keyframe describes.
posX, posY, posZ	The position of the origin of the camera's plane in the SynthesizedTexture's 3D world, in worldUnits.
rotX, rotY, rotZ	The rotation of the camera's plane about each of the 3D world's axes, in radians.
lFrameDiff	The difference between this KF's frame number and that of the previous KF.

3.1.8 Playback

During playback, the frames on the SynthesizedTexture's shared timeline are sequentially rendered to a shared canvas.

For each frame:

- The current position and orientation of the SynthesizedTexture's Camera is calculated, relative to the SynthesizedTexture's 3D world.
- The state of the texture of each of the SynthesizedTexture's objects per this frame is calculated, as described above.
- The bitmap image of each of the object textures is rendered, based on its state, and the state of the camera.
- The rendered bitmap images are projected to the shared canvas.
-

4 Coding and Bitstream

4.1 Overview

This section specifies the structure of the ST Bitstream and describes how it is decoded to SynthesizedTexture primitives, described in 3.1.

The ST Bitstream contains the ST primitives data in a highly re-organized arrangement, resulting in very high compression rates.

4.2 Global Input Bitstream and Decoding Context

The pseudo code below describes the main ST decoding procedure.

Additionally it introduces the global scope and initiation of the pseudo code sections in this chapter.

Names of global variables are prefixed with the 'g'.

4.2.1 Syntax

```
global context
{
  ...

  bitsStream gbsMain = new bitsStream(external_input_stream); // the main bit stream

  char signature = gbsMain.word(1);
  int  version   = gbsMain.word(2);

  Header          gHdr;         // the current decoded Header
  Scene           gScn;         // the current decoded Scene
  Camera          gCmr;         // the current decoded Camera
  Object          gObj;         // the current decoded Object
  Skeleton        gSkl;         // the current decoded Skeleton
  Animation       gAnm;         // the current decoded Animation
  Texture         gTxr;         // the current decoded Texture

  decHeaderBlock  gDecHdr;      // a Header decoder
  decSceneBlock   gDecScn;      // a Scene decoder
  decCamera       gDecCmr;      // a Camera decoder
  decSkeleton     gDecSkl;      // a Skeleton decoder
  decAnimation    gDecAnm;      // a Animation decoder
  decTextureBlock gDecTxr;      // a Texture decoder

  bitsStream      gbsTxr;       // the main sub-stream of the texture

  do
  {
    char blockType   = gbsMain.word(1);
    int  blockSize   = gbsMain.word(3);

    switch (blockType)
    {
      case 'H':
        int streamSize = gbsMain.word(4);
        gDecHdr = new decHeaderBlock(); // initaite Header decoding
        break;

      case 'S':
        gDecScn = new decSceneBlock();  // initaite Scene decoding
        break;

      case 'C':
        gDecObj = new decObjectBlock(); // initaite Object decoding
        break;

      case 'A':
        gDecTxr = new decTextureBlock();// initaite texture decoding
        break;
    }
  }
  while (!gbsMain.EndOfStream());

  ...
}
```

4.2.2 Semantics

Name	Description
gbsMain	This is the main and global to all program bitsStream (§4.11) of the decoder, from which all subStreams are then opened. gbsMain feeds from the external input stream provided to the decoder. gbsMain.EndOfStream() is assumed to return true iff the end of the external input stream is reached. Each time this 'function' is called it fetches the specified number of elements of the specified data type.
signature	The ST stream signature. Value must be 'V' (0x56).
version	Synthesized Texture version information.
blockType	Synthesized Texture Blocks are the top level sub-divisions of the Synthesized Texture bitstream. All blocks are preceded with a unique block-type-ID, read by the main loop of the decoder. Every block includes a blockType identifier in its first byte. 'H': Header - general information about the bitstream and its contents. 'S': Scene - a Scene description. 'C': Object - an Object description. 'A': Texture - a Texture description. Block types are described in the following sections. Within each Block, some of the data is further grouped and encoded in sub-structures of type bitsStream and toknStream which support the Synthesized Texture's high level of compression.
blockSize	The byte size of the following block.
streamSize	The byte size of the entire bitstream. Note this element is present ONLY before a Header block.
gHdr	The global decoded Header.
gDecHdr	The global decHeaderBlock class. Construction initiates header decoding into gHdr.
gScn	The global decoded Scene.
gDecScn	The global decSceneBlock class. Construction initiates Scene decoding into gScn.
gCmr	gScn's global decoded Camera Scenario.
gDecCmr	The global decCamera class associated with decSceneBlock. Construction occurs in decSceneBlock() and initiates Camera decoding into gCmr.
gObj	The current global decoded Object.
gDecObj	The global current decObjectBlock class. Construction initiates Object decoding into gObj.
gbsTxr	The main and global subStream of the texture.
gTxr	The current global decoded Texture.
GSkl	gTxr's global decoded Skeleton.
gAnm	gTxr's global decoded Animation.
gDecTxr	The global current decTextureBlock class. Construction initiates texture decoding into gTxr.
gDecSkl	The global decSkeleton class associated with decTextureBlock. Construction occurs in decSkeleton() and initiates skeleton decoding into gSkl.
gDecAnm	The global decAnimation class associated with decTextureBlock. Construction occurs in decAnimation() and initiates animation decoding into gAnm.

4.3 Header Block ('H') Decoding

A SynthesizedTexture Header block is identified by 'H' in its first byte.

4.3.1 decHeaderBlock()

4.3.1.1 Syntax

```
decHeaderBlock::decHeaderBlock()
{
  gHdr = new Header();  // the decoded header

  bool   bHasNumOfScenes = gbsMain.bit(1);
    bool   bHasTitle         = gbsMain.bit(1);
    bool   bHasArtist        = gbsMain.bit(1);
    bool   bHasDescription = gbsMain.bit(1);
    bool   bHasCopyright   = gbsMain.bit(1);
    bool   bHasDate          = gbsMain.bit(1);
    bool   bIsProtected     = gbsMain.bit(1);
    bool   bMore             = gbsMain.bit(1);
  if (bMore)
  {
    int    spare14           = gbsMain.bit(7);
    bool   bMore1            = gbsMain.bit(1);
  }

  int    numOfScenes = (bHasNumOfScenes   ? gbsMain.word(1));
  int    date        = (bHasDate          ? gbsMain.word(4));
  int    frameRate   = (bHasFrameRate     ? gbsMain.word(1));
  char[] title       = (bHasTitle         ? gbsMain.asciiz(16));
  char[] copyright   = (bHasCopyright     ? gbsMain.asciiz(16));
  char[] artist      = (bHasArtist        ? gbsMain.asciiz(16));
  char[] description = (bHasDescription   ? gbsMain.asciiz(256));
}
```

4.3.1.2 Semantics

Name	Description
bHasNumOfScenes	Is numOfScenes present?
bHasFrameRate	Is frameRate present?
bForceFrameRate	Should the player or it's user be allowed to modify the frame rate specified in frameRate?
spare3_6	Unused bits
bMore	Does another flags byte follow?
bHasTitle	Is title present?
bHasArtist	Is artist present?
bHasDescription	Is description present?
bHasCopyright	Is copyright present?
bHasDate	Is date present?
bIsProtected	Is playing the content in the bitstream protected by Digital Rights Management?
spare14	Unused bits
bMore1	Does another flags byte follow?
numOfScenes	Number of scenes in the bitstream. Present only if bHasNumOfScenes is true.
date	Creation date of the stream's contents. Present only if bHasDate is true.
frameRate	Recommended frame rate in frames/second in which the bitstream contents should be played. Present only if bHasFrameRate is true.
title	The stream's title. Present only if bHasTitle is true.
copyright	The stream's copyright string. Present only if is true.
artist	The stream's Artist. Present only if bHasCopyright is true.
description	A general description of the bitstream. Present only if bHasDescription is true.

4.4 Scene Block ('S') Decoding

4.4.1 decSceneBlock()

4.4.1.1 Syntax

```
decSceneBlock::decSceneBlock()
{
  gScn = new Scene();  // the decoded Scene

  bool   bHasName  = gbsMain.bit(1);
  bool   bHasScaledFrame             = gbsMain.bit(1);
  bool   bHasBackgroundColor = gbsMain.bit(1);
  bool   bHasCameraScenario      = gbsMain.bit(1);
  bool   bHasPreviewFrameNum = gbsMain.bit(1);
  bool   bHasFrameRate     = gbsMain.bit(1);
  bool   bspare        = gbsMain.bit(1);
  bool   bMore                 = gbsMain.bit(1);

  int    numOfObjects          = gbsMain.word(1);
  int    backgroundColor       = (bHasBackgroundColor ? gbsMain.word(3));
  char[] name                  = (bHasName            ? gbsMain.asciiz(63));
  int    numOfFrames           = gbsMain.bit(16);
  int    frameHeight           = gbsMain.bit(12);
  int    frameWidth            = gbsMain.bit(12);
  int    scaledFrameHeight     = (bHasScaledFrame     ? gbsMain.bit(12));
  int    scaledFrameWidth      = (bHasScaledFrame     ? gbsMain.bit(12));
  int    frameRate             = (bHasFrameRate       ? gbsMain.bit(8));
  int    previewFrameNum       = (bHasPreviewFrameNum ? gbsMain.bit(16));

  if (bHasCameraScenario)
  {
    gCmr = new Camera();          // gScn's Camera Scenario
    gDecCmr = new decCamera();    // initaite Camera decoding
  }
}
```

4.4.1.2 Semantics

Name	Description
bHasName	Is name present?
bHasScaledFrame	Are scaledFrameHeight and scaledFrameWidth present?
bHasBackgroundColor	Is backgroundColor present?
bHasCameraScenario	Is Camera Scenario Present?
bHasPreviewFrameNum	Is previewFrameNum present?
bHasFrameRate	is FrameRate present?
bspate	spare bit
bMore	Does another flags byte follow?
numOfObjects	The number of objects in this scene.
backgroundColor	The scene canvas's background color. Present only if bHasBackgroundColor is true.
name	The scene's name. Present only if bHasName is true.
previewFrameNum	Which frame did the author select as this scene's "preview" frame. Present only if bHasPreviewFrameNum is true. If absent, this scene has no preview frame.
numOfFrames	The number of frames in this scene.
frameHeight, frameWidth	The respective height and width of the scene's frame in pixels.
scaledFrameHeight, scaledFrameWidth	The respective height and width of the scene's scaled frame in pixels. Present only if bHasScaledFrame is true.
frameRate	Frame rate. Present only if bHasFrameRate is true.
cameraScenario	The camera scenario for this scene. If absent this scene is a "fixed camera" scene.

4.5 Object Block ('C') Decoding

An Object block is identified by 'C' in its first byte (objects were previously named "characters").

4.5.1 decObjectBlock()

4.5.1.1 Syntax

```
decObjectBlock::decObjectBlock()
{
  gObj = new Object();     // the decoded Object

  bool   bHasName         = gbsMain.bit(1);
  bool   bHasType         = gbsMain.bit(1);
  bool   bHasScaledHeight = gbsMain.bit(1);
  bool   bHasSkeleton     = gbsMain.bit(1);
  bool   bHasAnimation    = gbsMain.bit(1);
  bool   bFlip            = gbsMain.bit(1);
  bool   bText            = gbsMain.bit(1);
  bool   bMore            = gbsMain.bit(1);

  if (bMore)
  {
    bool   bLocked          = gbsMain.bit(1);
    bool   bMovable         = gbsMain.bit(1);
    bool   bFlipable        = gbsMain.bit(1);
    bool   bHasInteraction  = gbsMain.bit(1);
    bool   spare12_14       = gbsMain.bit(3);
    bool   bMore1           = gbsMain.bit(1);
  }

  char[] name         = (bHasName         ? gbsMain.asciiz(63));
  int    type         = (bHasType         ? gbsMain.bit(4));
  int    scaledHeight = (bHasScaledHeight ? gbsMain.bit(10));

  if (bHasSkeleton)
  {
    gSkl    = new Skeleton();
    gDecSkl = new decSkeleton();   // initaite Skeleton decoding
  }

  if (bHasAnimation)
  {
    gAnm    = new Animation();
    gDecAnm = new decAnimation();   // initaite Animation decoding
  }

  // Text               text;
  // Interaction        interaction;
}
```

4.5.1.2 Semantics

Name	Description
bHasName	Is name present?
bHasType	Is type present?
bHasScaledHeight	Is scaledHeight present?
bHasSkeleton	Is skeleton present?
bHasAnimation	Is animation present?
bFlip	Flip the object by 180?
bText	Are text properties present?
bMore	Does another flags byte follow?

bLocked	Can the object recieve interacion from the player.
bMovable	Is object moveable by the player?
bFlipable	Can object be rotated into the Z dimension by more than +/- 180 degrees by the player?
bHasInteraction	Is interaction information present?
spare12_14	Unused bits
bMore1	Does another flags byte follow?
name	The object's name.
type	The object's type. Present only if bHasType is true. • REGULAR (0x01) - This object was authored explicitly, i.e. the player does not need to generate it. • TEXT (0x02) - This object was authored to portray text. Text information serves only for future authoring of the object.
scaledHeight	The object's scaled height in pixels. Present only if bHasScaledHeight is true.
skeleton	The object's skeleton. Present only if bHasSkeleton is true.
animation	The object's animation. Present only if bHasAnimation is true.
text	The text properties of the object. Present only if bText is true and (type == TEXT).
interaction	The object's interaction rules. Present only bHasInteraction if is true.

4.6 Texture Block ('A') Decoding

A Texture block is identified by 'A' in its first byte.

4.6.1 Overview

This chapter describes the bitstream of the **Texture** and the process of decoding it to primitives, known as **Texture Decoding**.

A Texture block contains the data of a single Texture. It is comprised of a sequence of sub-streams, accessed during a series of specific decoding stages.

The target of the decoding process are **Texture Primitives**. §3.1.2 - The Texture describes the syntax and semantics of the target data structures of Texture Decoding.

4.6.2 decTextureBlock()

`Class decTextureBlock` reads and decodes all primitives comprising a single Texture.

Reading and decoding bitstream data to Texture primitives is done in 8 main **decoding stages**. Each stage is dedicated to decoding and re-constructing certain properties of texture primitives for the entire texture.

Texture decoding stages are described in detail in the sections following this one.

Each decoding stage accesses in parallel one or several **sub-streams** of the main bitstream. ST sub-streams group data elements that have been found empirically to reduce data size when properly aggregated, ordered, quantized, packed and compressed.

Texture sub-streams are described and listed in §4.11.

4.6.2.1 Syntax

```
decTextureBlock::decTextureBlock()
{
  bitsStream  gbsTxr(gbsMain);// open main sub-stream of the texture

  gTxr = new Texture();       // let gTxr be the decoded texture

  header                decHeader();

  locations             decLocations();

  curveGeometry         decLinesGeometry();

  subLayerIds           decSubLayerIds();

  areaColoring          decAreaColoring();

  lineColorProfiles     decLineColorProfiles();

  patches               decPatches();

  depths                decDepths();

}
```

4.6.2.2 Semantics

`Class decTextureBlock` reads and decodes all primitives comprising a single Texture.

Reading and decoding bitstream data to Texture primitives is done in 8 main **decoding stages**. Each stage is dedicated to decoding certain properties of texture primitives for the entire texture. Each decoding stage "SSS" is executed by a respective class "decSSS".

The Texture decoding stages are described in the sections following this one, as listed below.

Each decoding stage accesses in parallel one or several **sub-streams** of the main bitstream.

Within the main bitstream, sub-streams are ordered in a specific order, according to the first time they must be used in one of the decoding stages. Sub-streams are thus opened and read from the main bitstream immediately prior to the decoding stage that first uses them:

All Texture sub-streams extend the `toknStream` or `bitsStream` classes described and listed in §4.11.

Decoding Stage	Section
a. decHeader	**4.6.3**
• information about the texture as a whole e.g. texture dimensions.	
• information about the texture's coding and bitstream e.g. quantization levels.	
• Geometric information about the texture's Layers (LYs).	
b. decLocations	**4.6.4**
• The 2D locations of all Terminal Points (TPs) in the texture.	
• The 2D locations of the center points of all Patches (PA) in the texture.	
c. decLinesGeometry	**4.6.5**
• The geometric shape of all Line Segments (LS) in the texture.	
• The Line Type of all Lines (LNs) in the texture.	
• The width components of all Line Color Profiles (LCs) in the texture.	
• How Lines (LNs) in the texture are connected.	
d. decLayersIds	**4.6.6**
• Association of texture primitives with Layers.	
• Attributing contour type to LNs.	
e. decAreaColoring	**4.6.7**
• The location and color of all Area Color Points (ACs) in the texture.	
• The margin color values of all Line Color Profiles (LCs) in the texture.	
f. decLineColorProfiles	**4.6.8**
• All remaining properties of the Line Color Profiles (LCs) in the texture.	
g. decPatch	**4.6.9**
• All remaining properties of the geometry and coloring of the patches (PAs) in the texture.	
h. decDepths	**4.6.10**
• All depths of PAs, ACs and LCs, in case they were not previously decoded globally.	

4.6.3 Texture Header

4.6.3.1 decHeader

4.6.3.1.1 Syntax

```
class decHeader::decHeader()
{
  // texture coding flags:
  bool bHasDepth       = gbsTxr.bit(1);
  bool bHasContour     = gbsTxr.bit(1);
  bool bHasRidges      = gbsTxr.bit(1);
  bool bHasPatches     = gbsTxr.bit(1);
  bool bHasDepthDeltas = gbsTxr.bit(1);
  bool spare5_6        = gbsTxr.bit(2);
  bool bMore           = gbsTxr.bit(1);

  gTxr.width           = gbsTxr.bit(16);
  gTxr.height          = gbsTxr.bit(16);
  int  cellSize        = gbsTxr.bit(8);
  int  qLvl            = gbsTxr.bit(8);

  if (bHasDepth)
  {
    gTxr.worldUnit = gbsTxr.bit(16);

    if (gbsTxr.bit(1) == 1)
    {
      gTxr.LosOffsetX = gbsTxr.bit(16);
      gTxr.LosOffsetY = gbsTxr.bit(16);
    }

    decLayers();
    decSubLayersDepath();
  }
}
```

4.6.3.1.2 Semantics

Class decHeader reads global information regarding:

- The texture as a whole e.g. texture dimensions.
- The texture's coding and bitstream parameters e.g. quantization levels.
- Layers' information later used to associate and orientate texture primitives relative Bones.

Name	Description
bHasDepth	Does the coded texture have depth
bHasContour	Does the coded texture have a contour line (that is not its bounding rectangle) ?
bHasRidges	Does the coded texture include RIDGE lines ?
bHasPatches	Does the coded texture include Patches (PAs) ?
bHasDepthDeltas	Do primitives in the coded texture have depth corrections (deltas) relative to their layers? Default: false, i.e. primitives' depth is derived from their layer's depth information.
spare5_6	Unused bits
BMore	Does another flags byte follow?
CellSize	The size of the top-level cells used for encoding the texture's occupancy grid, in pixels. This is typically 8, indicating cells of 8x8 pixels.
qLvl	The quantization level used for encoding the texture – reflects level of detail and sampling resolution. This is a 0..4 index into quantization tables, see §4.6.9.1.2.

4.6.3.2 decLayers()

4.6.3.2.1 Syntax

```
decHeader::decLayers()
{
  gTxr.nLY = gbsTxr.bit(8);

  int Xbits = ilog2(height/cellSize);
  int Ybits = ilog2(width /cellSize);

  for (int lyi = 0; lyi < gTxr.nLY; lyi++)
  {
    gTxr.aLY[lyi].nSL = gbsTxr.bit(4);
    gTxr.nSL += gTxr.aLY[lyi].nSL;
    for each SL sl in gTxr.aLY[lyi]
      sl.iLY = lyi;

    gTxr.aLY[lyi].X0 = gbsTxr.bit(Xbits) * cellSize;
    gTxr.aLY[lyi].X1 = gbsTxr.bit(Xbits) * cellSize;
    gTxr.aLY[lyi].Y0 = gbsTxr.bit(Ybits) * cellSize;
    gTxr.aLY[lyi].Y1 = gbsTxr.bit(Ybits) * cellSize;

    gTxr.aLY[lyi].bHasDepthDeltas = gbsTxr.bit(1);

    gTxr.aLY[lyi].name = gbsTxr.asciz(64);
  }
}
```

4.6.3.2.2 Semantics

Name	Description
Xbits, Ybits	Number of bits to be used in reading X and Y coordinates of layers' bounding rectangles.

4.6.3.3 decSubLayersDepth()

4.6.3.3.1 Syntax

```
decHeader::decSubLayersDepth()
{
  int nBitsForDistance = gbsTxr.bit(6);

  for each SL sl in gTxrgTxr
  {
    sl.surfaceType = gbsTxr.bit(4);
    sl.bOrthogonal = gbsTxr.bit(1);

    if (bOrthogonal)
      sl.orient = point3D(0, 0, 1.0);
    else
    {

      sl.orient.x = gbsTxr.float(qVecDist, qVecDist.bits());
      sl.orient.y = gbsTxr.float(qVecDist, qVecDist.bits());
      sl.orient.z = gbsTxr.float(qVecDist, qVecDist.bits());


      // normalize orient:
      float tmp = sqrt(sl.orient.x^2 + sl.orient.y^2 + sl.orient.z^2);
      sl.orient.x /= tmp;
      sl.orient.y /= tmp;
      sl.orient.z /= tmp;
    }

    sl.dist = gbsTxr.float(qVecDist, nBitsForDistance);
  }
}
```

4.6.3.3.2 Semantics

Name	Description
nBitsForDistance	Specifies the number of bits to read for the distance float values

4.6.4 Locations

4.6.4.1 decLocations()

4.6.4.1.1 Syntax

```
class decLocations::decLocations()
{
  // uses:
  bitsStream bsCntr    (gbsMain);
  toknStream ts4x1cell (gbsMain);
  toknStream tsLocType (gbsMain);

  int flags        = gbsTxr.bit(8);    // reserved bit
  int cellSize     = gbsTxr.bit(8);

  qCord.min = -cellSize/2;
  qCord.max =  cellSize/2;

  for each cell8x8 in the texture  // for each group of 8X8 cells in aCell[]
    decCell8X8(cell8x8)
}
```

```
decLocations::decCell8X8(cell8x8)
{
  int bMultiple = 0;

  if (bsCntr.bit(1) == 1)
  {
    bMultiple = gbsTxr.bit(1);

    for each cell4x4 in cell8x8  // for each group of 4X4 cells in cell8x8
      decCell4x4(cell4x4, bMultiple);
  }
}
```

```
decLocations::decCell4x4(cell4x4, bMultiple)
{
  if (bsCntr.bit(1) == 1)
  {
    if (bMultiple == 0)
      bMultiple = bsCntr.bit(1);

    for each cell2x2 in cell4x4  // for each group of 2X2 cells in cell4x4
      decCell2x2(cell2x2, bMultiple);
}
```

```
decLocations::decCell2x2(cell2x2, bMultiple)
{
  if (bsCntr.bit(1) == 1)
  {
    int 4x1cell = ts4x1cell.next();

    for each cell1x1 in cell2x2
      if (corresponding bit in 4x1cell == 1)
        decCell1x1(cell1x1, bMultiple);
}
```

```
decLocations::decCell1x1(cell, bMultiple)
{
  do
  {
     int type = ssLocType.next();
     switch(type)
     {
        case 0:
          aPA[nPA].x = cell.x + dequant(qCoord, bsCntr.bit(qCoordBits));
          aPA[nPA].y = cell.y + dequant(qCoord, bsCntr.bit(qCoordBits));
          nPA++;
          break;

        case default:
          aTP[nTP].x  = cell.x + dequant(qCoord, bsCntr.bit(qCoordBits));
          aTP[nTP].y  = cell.y + dequant(qCoord, bsCntr.bit(qCoordBits));
          aTP[nTP].junctionType = type - 1;
          nTP++;
          break;
    }
  }
  while(bMultiple && bsCntr.bit(1) == 1);
}
```

4.6.4.1.2 Semantics

Class decLocations reads and decodes:

- The 2D locations of the Terminal Points (TPs), which define the start and end points of all the texture's Lines (LNs).
- The 2D locations of the centers of all Patches (PAs) in the texture.

Locations of **centers** (the geometric centers of TPs and PAs) in the Texture are encoded using the following procedure:

- The area of the Texture is divided into cells of cellSize x cellSize pixels.
- An **occupancy matrix** marks the number of centers in each cell.
- The occupancy matrix is encoded into a stream using the **partial quad-tree** algorithm.
- the partial quad-tree algorithm establishes a **reference order** to all centers.
- The (x, y) offset of each center from the center of it's cell is recorded.

Name	Description
Type	A location may be TerminalPoint or Patch

4.6.5 Line Geometry

4.6.5.1 decLinesGeometry()

4.6.5.1.1 Syntax

```
void decLinesGeometry::decLinesGeometry()
{
  // uses:
  toknStream tsLsCnt (gbsMain);
  toknStream tsLsHgt (gbsMain);
  toknStream tsLsVec (gbsMain);
  toknStream tsEndTp (gbsMain);

  flags = gbsTxr.bit(8);  // reserved bit

  decTerminalPointTopology();

  decBranchingLines();

  decLineWidths();  // RIDGE and EDGE

  decParallelLines();
}
```

4.6.5.1.2 Semantics

Class decLinesGeometry reads and decodes:

- The geometric shape of Line Segments (LS) in the texture.
- The Line Type of Lines (LN) in the texture.
- The width components of Line Color Profiles (LC) in the texture.
- How Lines (LN) in the texture are connected.

4.6.5.2 decTerminalPointTopology()

4.6.5.2.1 Syntax

```
void decLinesGeometry::decTerminalPointTopology()
{
  for each TP tp in gTxr
  {
    if (tp.JunctionType == SPLITTING)
      decSplittingTopology(tp);
    else
    {
      tp.nBR = gbsTxr.bit(2);

      for each BR br in tp
      {
        br.bOut = true
        br.iLnType = tp.JunctionType;
      }
    }
  }
}
```

4.6.5.2.2 Semantics

This class decodes the type of the Treminal Point. It it is of splitting nature, it calls for decoding all the lines ending at this point.

4.6.5.3 decBranchingLines()

4.6.5.3.1 Syntax

```
void decLinesGeometry::decBranchingLines()
{
  for each tp in gTxr.aTP[]
  {
    for each BR br in tp
      if (br.bOut)
        decBranchingLine(tp, br);
  }
}
```

4.6.5.4 decBranchingLine()

4.6.5.4.1 Syntax

```
void decLinesGeometry::decBranchingLine(TP tpBeg, BR brBeg)
{
  LN ln;

  ln.iLN = nLN;
  ln.iLnType = brBeg.iLnType;
  ln.nLS = tsLsCnt.next() + 1;

  int xTotal = tpBeg.x;
  int yTotal = tpBeg.y;
  int xBase = 0;
  int yBase = 0;

  for each LS ls in ln
  {
    ls.hgt = dequantZ(qLsHgt, tsLsHgt.next());

    if (ls is NOT the last LS in ln)
    {
      ls.x = dequantZ(qLsVec, tsLsVec.next()) + xBase;
      ls.y = dequantZ(qLsVec, tsLsVec.next()) + yBase;
      xBase = ls.x;
      yBase = ls.y;

      xTotal += ls.x;
      yTotal += ls.y;
    }
  }

  brBeg.iLN = ln.iLN;
  ln.iTpBeg = tpBeg.iTP;
  ln.iTpEnd = decLineEndTp(ln.iLnType, xTotal, yTotal);

  TP  tpEnd  = gTxr.aTP[ln.iTpEnd];
  int iBR;

  if (tpEnd.junctionType == SPLITTING)  // only BR[1] and BR[2] are possible
    iBR = gsbTxr.bit(1) + 1;
  else
    iBR = tpEnd.nBR++;                  // add a new in-branch

  BR brEnd = tpEnd.aBR[iBR];
  brEnd.iLN = ln.iLN;
  brEnd.bOut = false;
  brEnd.iLnType = ln.iLnType;

  let ls be the last LS in ln;
  ls.x = tpEnd.x - xTotal;
  ls.y = tpEnd.y - yTotal;

  gTxr.aLN[gTxr.nLN++] = ln;            // add ln
}
```

4.6.5.4.2 Semantics

This class decodes and computes the line geometry. Coordinates are specified as offsets from previous coordinates.

4.6.5.5 decEndTP()

4.6.5.5.1 Syntax

```
int decLinesGeometry::decLineEndTp(lineType, x, y)
{
  let cell0 be the cell in which (x, y) resides.

  let vicinity be a NxN cell vicinity around cell0 where
  {
    the cells in vicinity are indexed 0.. N^2-1 from vicinity's top-left;
    cell0 is in the central row and column of vicinity.
    N = 11;
  }

  int iCell = tsEndTp.next();  // the index of the cell in vicinity

  let aTpsInCell[] be the list of TPi's in cell[iCell] where
  {
    only TPs whose junctionType is SPLITTING or lineType are in the list;
    TPs in the list are ordered by their order in gTxr.aTP[];
  }

  let nTpsInCell be the number of elements in aTpsInCell;

  if (nTpsInCell == 1)
    return aTpsInCell[0]; // the single element in aTpsInCell
  else
  {
    int nBitsFor_iTP = ilog2(nTpsInCell - 1);
    int iTP = gsbTxr.bit(nBitsFor_iTP) + 1;
    return aTpsInCell[iTP];
  }
}
```

4.6.5.5.2 Semantics

Return the index of the ending TP of the Line starting at (x, y).

0	1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20	21
22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43
44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	**cell0**	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75
76	77	78	79	80	81	82	83	84	85	86
87	88	89	90	91	92	93	94	95	96	97
98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119

Figure 34 — The vicinity grid (N=11) of (x, y), and the indexing of its cells.

4.6.5.6 decLineWidths()

4.6.5.6.1 Syntax

```
void decLinesGeometry::decLineWidths()
{
  for each LN of iLnType RIDGE or EDGE in gTxr
  {
    let {p0, p1, …, pn} be the separating LCs on LN.
    let {q0, q1, …, qm} be the subset of {p0, p1, …, pn} so that
    {
      q0 = p0;
      qi = the next pj so (quasiDistance(pj, qi-1) > qLineSampleDistance.step);
      qm = pn;
    }

    for each qi
    {
      if (ln.iLnType == RIDGE)
      {
        qi.widthLft = dequant(qSepW, tsSepW.next());
        qi.widthRgt = dequant(qSepW, tsSepW.next());
      }
      else // EDGE
      {
        qi.widthLft = dequant(qSepW, tsSepW.next());
        qi.widthRgt = qi.widthLft;
      }
    }

    for each segment (qi-1, qi)
    for each p between qi-1 and qi
    {
      int dist = quasiDistance(p, qi-1) / quasiDistance(qi, qi-1);
      p.widthLft = widthInterpol(qi-1.widthLft, qi.widthLft, dist);
      p.widthRgt = widthInterpol(qi-1.widthRgt, qi.widthRgt, dist);
    }
  }
}
```

4.6.5.6.2 Semantics

This class determines the width of the line according to its type: Edge or Ridge.

4.6.5.7 decSplittingTopology()

4.6.5.7.1 Syntax

```
void decLinesGeometry::decSplittingTopology(TP tp)
{
  tp.nBR = 3;
  tp.aBR[0].iLnType = RIDGE;
  tp.aBR[0].bOut = gbsTxr.bit(1);

  for (int bri=1; bri<2; bri++) // for each BR in tp.aBR[1..2]
  {
    iSplitType = gbsTxr.bit(2);
    switch(iSplitType)
    {
      case 0:
        br.iLnType = ABSENT;
        br.bOut = false;
        break;
      case 1:
        br.iLnType = PARALLEL;
        br.bOut = false;
        break;
      case 2:
        br.iLnType = EDGE;
        br.bOut = false;
        break;
      case 3:
        br.iLnType = EDGE;
        br.bOut = true;
        break;
    }
  }
}
```

4.6.5.7.2 Semantics

This class decodes the nature of the splitting terminal point. Based on the topology the type of the branching lines is set.

4.6.5.8 decParallelLines()

4.6.5.8.1 Syntax

```
void decLinesGeometry::decParallelLines()
{
  for each SPLITTING TP tp in aTxr
  {
    for each BR br in tp
      if (br.iLineType == PARALLEL)
        br.iLN = decParallelLine(tp.x, tp.y);
  }
}
```

4.6.5.9 decParallelLine()

4.6.5.9.1 Syntax

```
int decLinesGeometry::decParallelLine(x, y)
{
  let cell0 be the cell in which (x, y) resides.

  let vicinity be a NxN cell vicinity around cell0 where:
  {
    cell0 is in the central row and column of vicinity.
    the cells in vicinity are indexed 0.. N^2-1, starting from vicinity's top-left;
    N = 11;
  }

  int iCell = tsEndTp.next();  // the index of the cell in vicinity

  let aLNsInCell[] be the list of LNi's where for each LN[LNi] ln in the list where:
  {
    some LS ls on ln fullfills (ls.vX, ls.vY) is in cell[iCell]; // ln "passes" in
cell[iCell]
    ln.iLnType==RIDGE;
    LNi's in the list are ordered by their order in gTxr.aLN[];
  }

  let nLnInCell be the number of elements in aLnInCell;

  if (nLnsInCell == 1)
    return aLnInCell[0]; // the single element in aLnInCell
  else
  {
    int nBitsFor_iLN = ilog2(nLnInCell - 1);
    int iLN = gsbTxr.bit(nBitsFor_iLN);
    return aLnInCell[iLN];
  }
}
```

4.6.5.9.2 Semantics

Return the index of the parallel Line branching at (x, y).

4.6.6 Layer Ids

4.6.6.1 decSubLayerIds()

4.6.6.1.1 Syntax

```
decSubLayerIds::decSubLayerIds()
{
  if(gTxr.nLY == 1 && !gTxr.bHasCountor)
  {
    for all PAs and LNs in gTxr:
      let iSL be 0;
      let iContourType be 0;
    return;
  }

  bool bHasMoreThanOneSL;
  bool bHasContourLine;
  int  iSL;

  int flags = gbsTxr.bit(8);
  for each cell2x2 in the texture with Lines or Patches
  {
    if (gTxr.nSL == 1)
    {
      bHasMoreThanOneSL = false;
      iLS = 0;
    }
    else
    {
      bHasMoreThanOneSL = gbsTxr.bit(1);
      if (!bHasMoreThanOneSL)   // i.e. has one
        iSL = tsSbLyId.next();
    }

    bHasContourLine = gbsTxr.bit(1);

    for each PA pa whose center is in cell2x2
      pa.iLS = NextSubLayerId(bHasMoreThanOneSL, iSL);

    for each LN ln starting in cell2x2
    {
        ln.iSlLft = ln.iSlRgt = NextSubLayerId(bHasMoreThanOneSL, iLS);
        ln.iContourType = 0;

        if (ln.iLnType == RIDGE || ln.iLnType == EDGE)
        {
          ln.iContourType = NextContourFlag(bHasContourLine);

          if (ln.iContourType == 0x04)
            iSlRgt = tsSbLyId.next();
        }
    }
  }
}
```

```
decSubLayerIds::nextSubLayerId(bool bHasMoreThanOneSL, int iSL)
{
  if (!bHasMoreThanOneSL)
    return(iSL);

  return(tsSbLyId.next();
}
```

```
decSubLayerIds::nextContourFlag(bool bHasContourLine)
{
  if (!bHasContourLine)
    return(0);

  return(tsSepFlg.next();
}
```

4.6.6.1.2 Semantics

Class decSubLayerIds reads and decodes:

- iSL's, used to associate LNs and PAs with a Layer and Sub-layer.
- Contour type, used to determine which LNs are "contours" of the texture and of sub-layers.

4.6.7 Area Coloring

4.6.7.1 decAreaColoring()

4.6.7.1.1 Syntax

```
decAreaColoring::decAreaColoring()
{
  //uses:
  bitsStream bsAc         (gbsMain);
  toknStream tsAcAbsY     (gbsMain);
  toknStream tsAcAbsCrCb  (gbsMain);
  toknStream tsAcRelY     (gbsMain);
  toknStream tsAcRelCrCb  (gbsMain);

  int flags    = gbsTxr.bit(8);     // reserved bit
  int cellSize = gbsTxr.bit(8);

  for each layer in the texture
  {
     decPointsLoaction();
     decPointsColor();
     decSepLinesMidAndCenterColors();
  }
}
```

4.6.7.1.2 Semantics

Class decAreaColoring reads and decodes:

- The location and color of Area Color Points (ACs).
- The far (colorFarLft, colorFarRgt) and central (colorCenter) color values of Line Color Profiles (LCs).

4.6.7.2 decPointsLoaction()

4.6.7.2.1 Syntax

```
decAreaColoring::decPointsLoaction()
{
  for each cell4x4 in the layer  // for each group of 4X4 cells in the layer
    decPointsLocationCell4x4(cell4x4)
}
```

```
decAreaColoring::decPointsLocationCell4x4(cell4x4)
{
  if (bsAc.bit(1) == 1)
  {
    for each cell2x2 in cell4x4  // for each group of 2X2 cells in cell4x4
      decPointsLocationCell2x2(cell2x2)
  }
}
```

```
decAreaColoring::decPointsLocationCell2x2(cell2x2)
{
  if (bsAc.bit(1) == 1)
  {
    for each cell1x1 in cell2x2  // for each group of 1X1 cells in cell2x2
      decPointsLocationCell1x1(cell1x1)
  }
}
```

```
decAreaColoring::decPointsLocationCell1x1(cell1x1)
{
  if (bsAc.bit(1) == 1)
  {
        aAC[nAC].x  = cell1x1.x;
        aAC[nAC].y  = cell1x1.y;

        aAC[nAC].iSL = iSL;
        nAC++;
  }
}
```

4.6.7.2.2 Semantics

These set of classes decodes the position of Area Color points (AC) of the layer. The points are encoded based on a quad-tree topology.

4.6.7.3 decPointsColor()

4.6.7.3.1 Syntax

```
decAreaColoring::decPointsColor()
{
  color nbrsAverage;

  for each cell C in gTxr containing AC point
  {
    if (PneighborsAverageColor(C, &nbrsAverage) == 1)
    {
      Ac.y   = nbrsAverage.y   + dequantZ(qAcY,    ssAcRelY.next());
      Ac.cr  = nbrsAverage.cr  + dequantZ(qAcCrCb, ssAcRelCrCb.next());
      Ac.cb  = nbrsAverage.cb  + dequantZ(qAcCrCb, ssAcRelCrCb.next());
    }
    else
    {
      Ac.y   = dequant(qAcY,   ssAcAbsY.next());
      Ac.cr  = dequant(qAcCr,  ssAcAbsCrCb.next());
      Ac.cb  = dequant(qAcCb,  ssAcAbs1CrCb.next());
    }
  }

  for each AC ac in gTxr.aAC[]
    ac.color.toRGB();
}
```

4.6.7.3.2 Semantics

The color of a point is computed form the average color of the neighboring point.

Name	Description
nbrsAverage	This variable holds the color average of the neighboring Area Color points

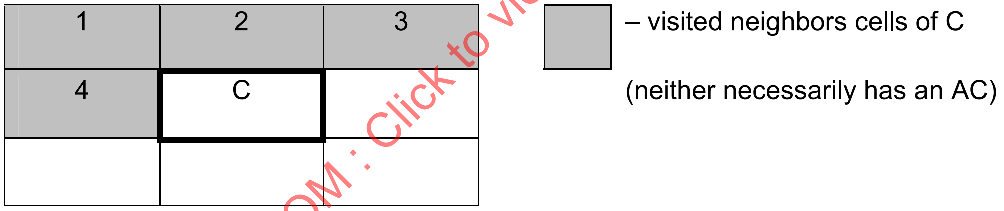
4.6.7.4 PneighborsAverageColor()

4.6.7.4.1 Syntax

```
bool PneighborsAverageColor(Cell C, color &nbrsAverage)
{
  nbrsAverage = { 0, 0, 0 };
  int  nAC = 0;

  for each AC in Pneighbor(C)
  {
    nbrsAverage.y += AC.y;
    nbrsAverage.cr += AC.cr;
    nbrsAverage.cb += AC.cb;
    nAC++;
  }

  if (nAC == 0) return(0);

  nbrsAverage.y  /= nAC;
  nbrsAverage.cr /= nAC;
  nbrsAverage.cb /= nAC;

  return(1);
}
```

4.6.7.4.2 Semantics

Pneighbors (previous neighbors) to a cell C are numbered as follows:


1
2
3
4
C
– visited neighbors cells of C
(neither necessarily has an AC)

4.6.7.5 decSepLinesMidAndCenterColors()

4.6.7.5.1 Syntax

```
decAreaColoring::decSepLinesMidAndCenterColors()
{
  for each LN of iLnType RIDGE or EDGE in gTxr
  {
    let {p0, p1, …, pn} be the separating LCs on LN.
    let {q0, q1, …, qm} be the subset of {p0, p1, …, pn} so that
    {
      q0 = p0;
      qi = the next pj so (quasiDistance(pj, $q_{i-1}$) > qLineSampleDistance.step);
      qm = pn;
    }

    for each qi
    for each parameter colorPrm in {qi.colorFarRgt, qi.colorFarLft, qi.colorCenter}
    {
      if (i == 0)
      {
        qi.colorPrm.y   = dequant(qAcY,     tsAcAbsY.next());
        qi.colorPrm.cr  = dequant(qAcY,     tsAcAbsCrCb.next());
        qi.colorPrm.cb  = dequant(qAcCrCb,  tsAcAbsCrCb.next());
      }
      else
      {
        q_i.colorPrm.y  = q_{i-1}.colorPrm.y  + dequantZ(qAcY,     tsAcRelY.next());
        q_i.colorPrm.cr = q_{i-1}.colorPrm.cr + dequantZ(qAcY,     tsAcRelCrCb.next());
        q_i.colorPrm.cb = q_{i-1}.colorPrm.cb + dequantZ(qAcCrCb,  tsAcRelCrCb.next());
      }
    }

    for each segment (q_{i-1}, q_i)
    for each p between q_{i-1} and q_i
    {
      int dist = quasiDistance(p, q_{i-1}) / quasiDistance(q_i, q_{i-1});
      p.colorFarRgt = colorInterpol(q_{i-1}.colorFarRgt, q_i.colorFarRgt, dist);
      p.colorFarLft = colorInterpol(q_{i-1}.colorFarLft, q_i.colorFarLft, dist);
      p.colorCenter = colorInterpol(q_{i-1}.colorCenter,  q_i.colorCenter,  dist);
    }

    for each LC lc on LN
      for each parameter colorPrm in {lc.colorFarRgt, lc.colorFarLft, lc.colorCenter}
        lc.colorPrm.toRGB();
  }
}
```

4.6.7.5.2 Semantics

The YcbCr color components of separating lines are decoded in this class

4.6.8 Line Color Profiles

4.6.8.1 decLineColorProfiles()

4.6.8.1.1 Syntax

```
class decLineColorProfiles::decLineColorProfiles()
{
  toknStream tsSepY         (gbsMain);
  toknStream tsSepCrCb      (gbsMain);
  toknStream tsSepW         (gbsMain);
  toknStream tsNsepDifY     (gbsMain);
  toknStream tsNsepDifCrCb  (gbsMain);
  toknStream tsNsepCrv        (gbsMain);

  int flags = gbsTxr.bit(8);      // reserved bit

  decSepartingColorProfiles();
  decStripeColorProfiles();
}
```

4.6.8.1.2 Semantics

Class decLineColorProfiles reads and decodes:

- The remaining Line Color Profile (LC) data that was not encoded in previous decoding stages.