Abstractions: From C to D

Andreas Zwinkau
KIT

TOPCONF
Duesseldorf

Abstractions: From C to D

Pepper:
Hello everybody! I'm Pepper.

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy

Actions: From C to D

Pepper:
What is the best programming language in the world?

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy

Why abstract?

Why abstract?

Edsger Wybe Dijkstra:
The effective exploitation of his powers of abstraction
must be regarded as one of the most vital activities of a competent programmer.

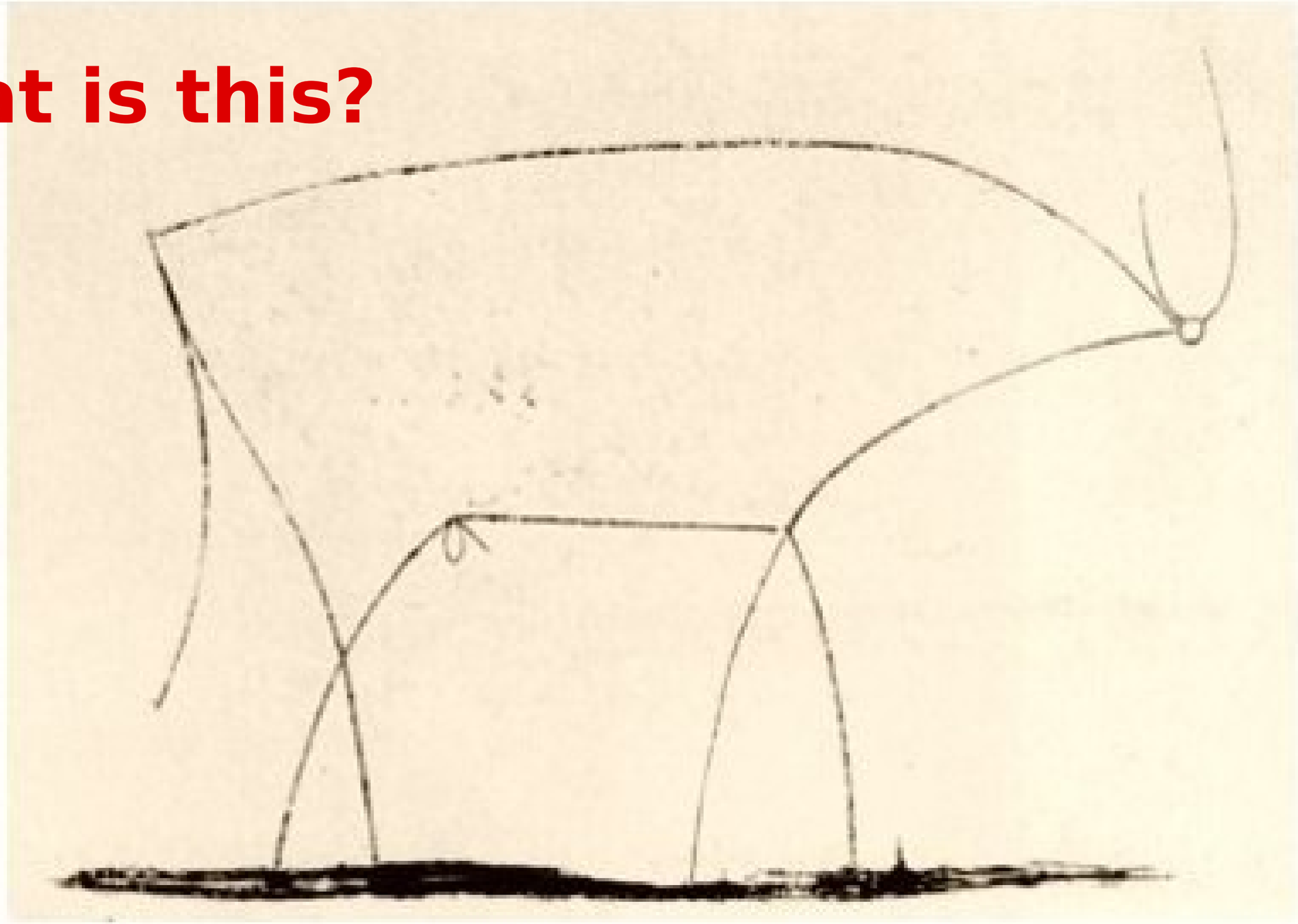
image from Wikipedia

bstract?

Pepper:
It‘s like potions for witches?

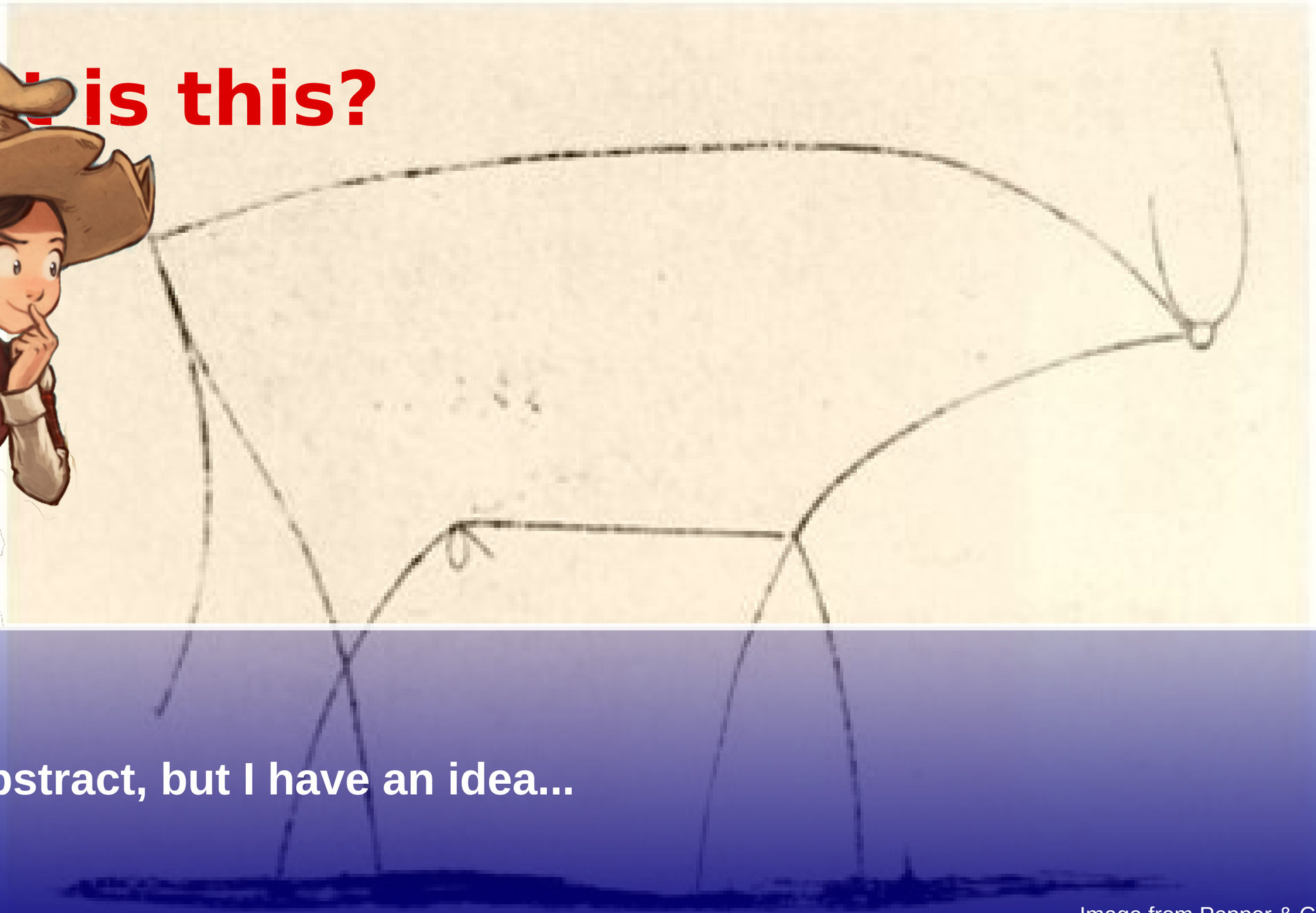
Image from Pepper & Carrot by David Revoy

What is this?



TOPCONF
Duesseldorf

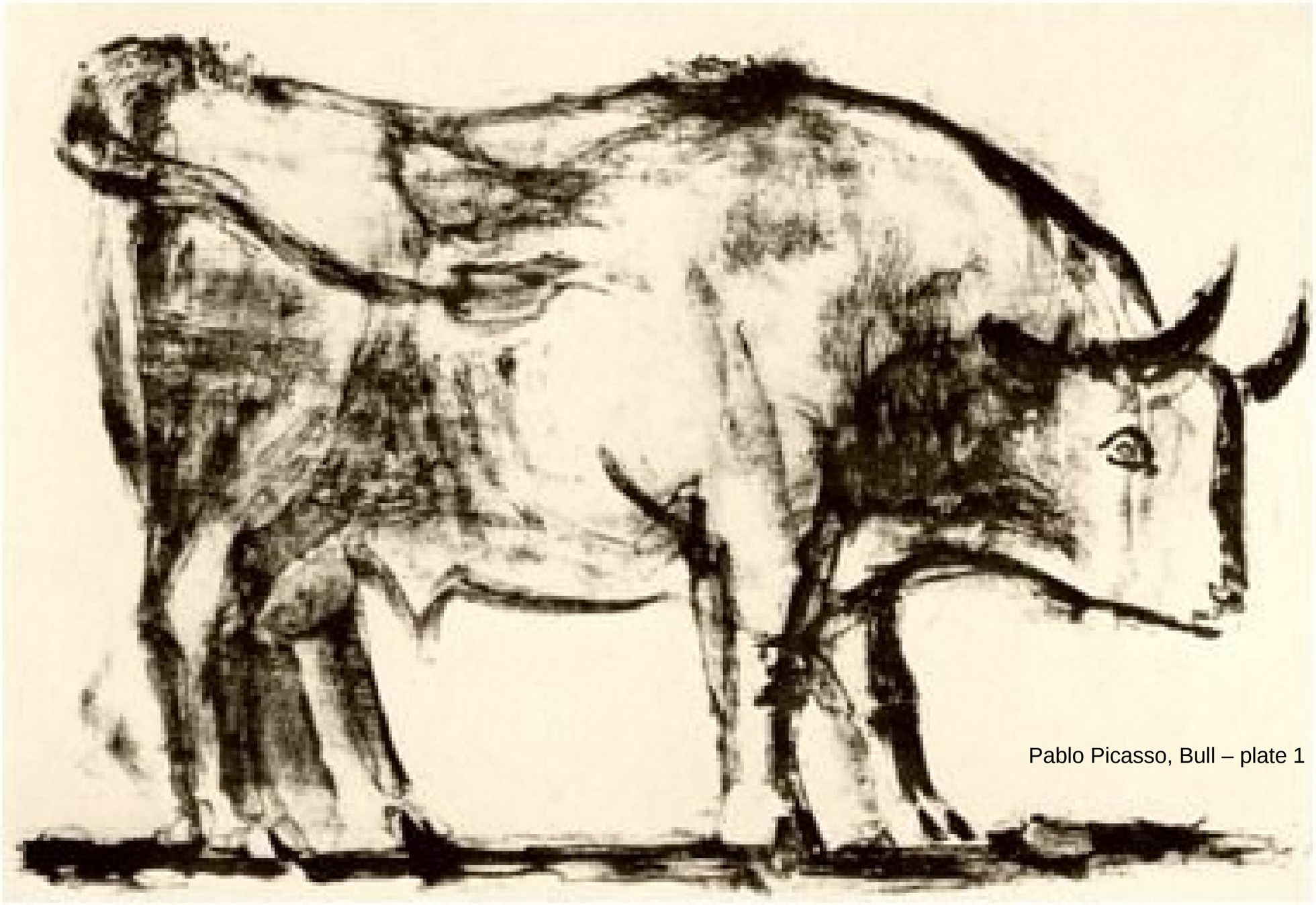
What is this?



Pepper:
Pretty abstract, but I have an idea...

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy



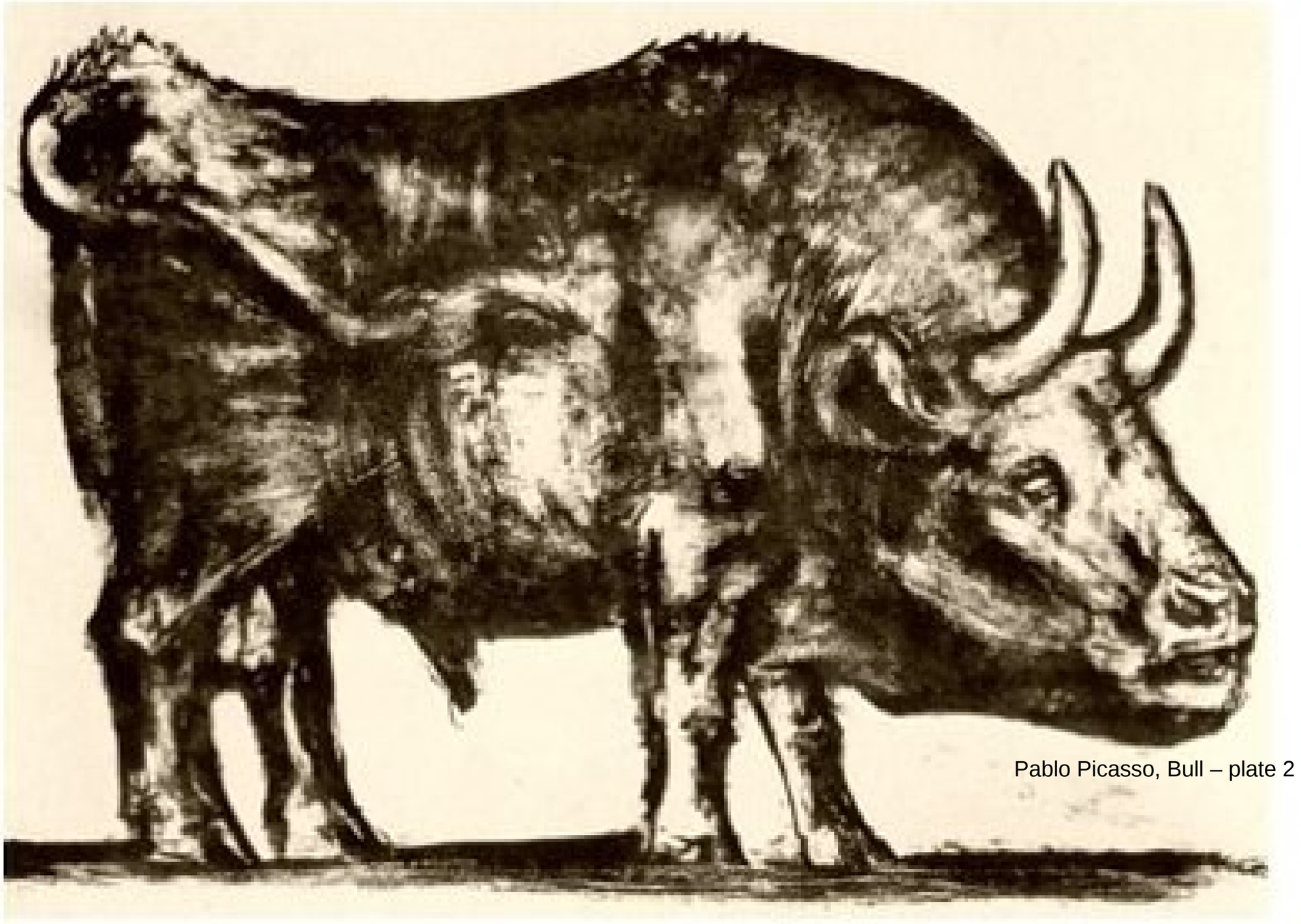
Pablo Picasso, Bull – plate 1

Pepper:
I knew it. I knew it. I knew it.

Pablo Picasso, Bull – plate 1

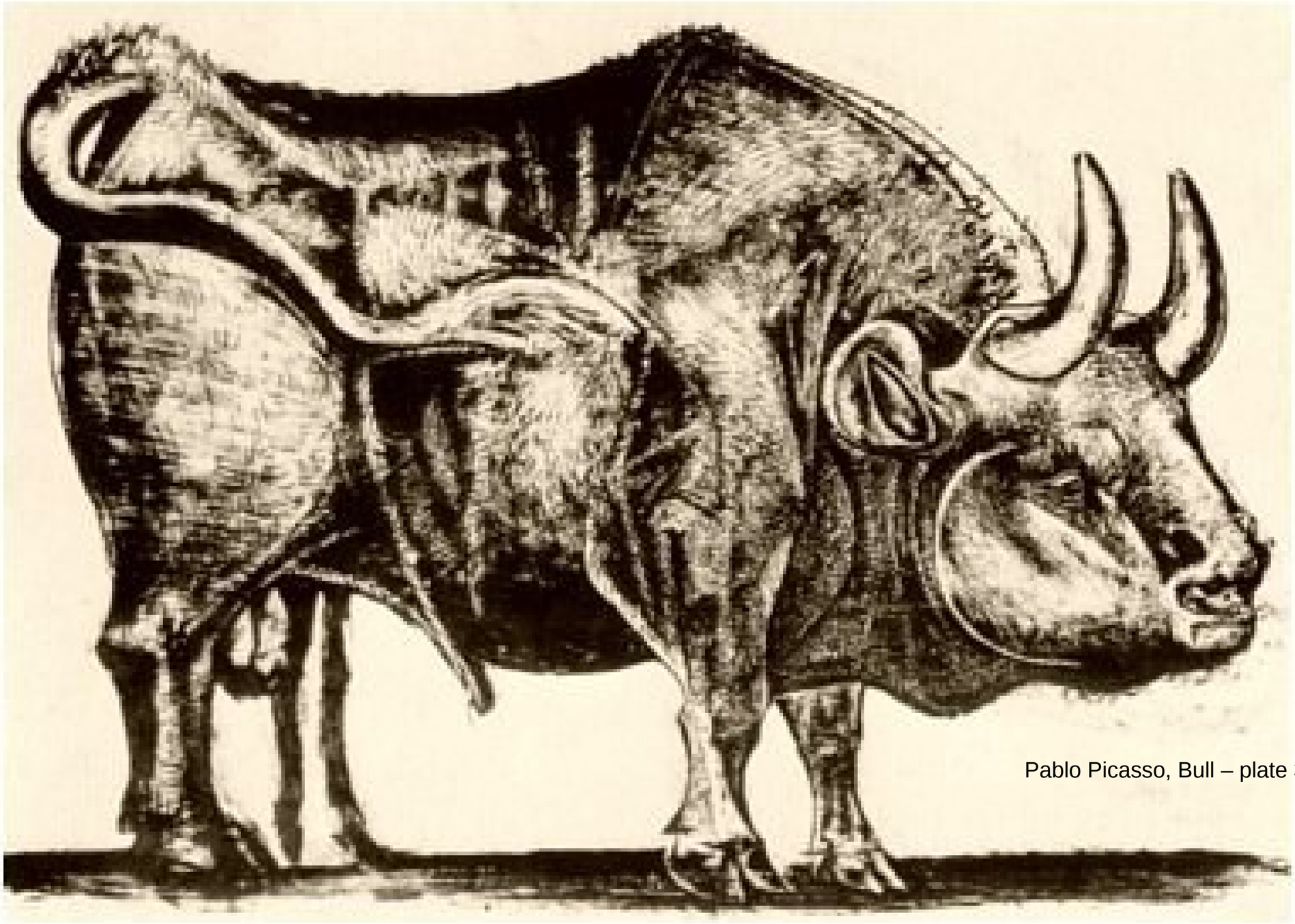
TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy



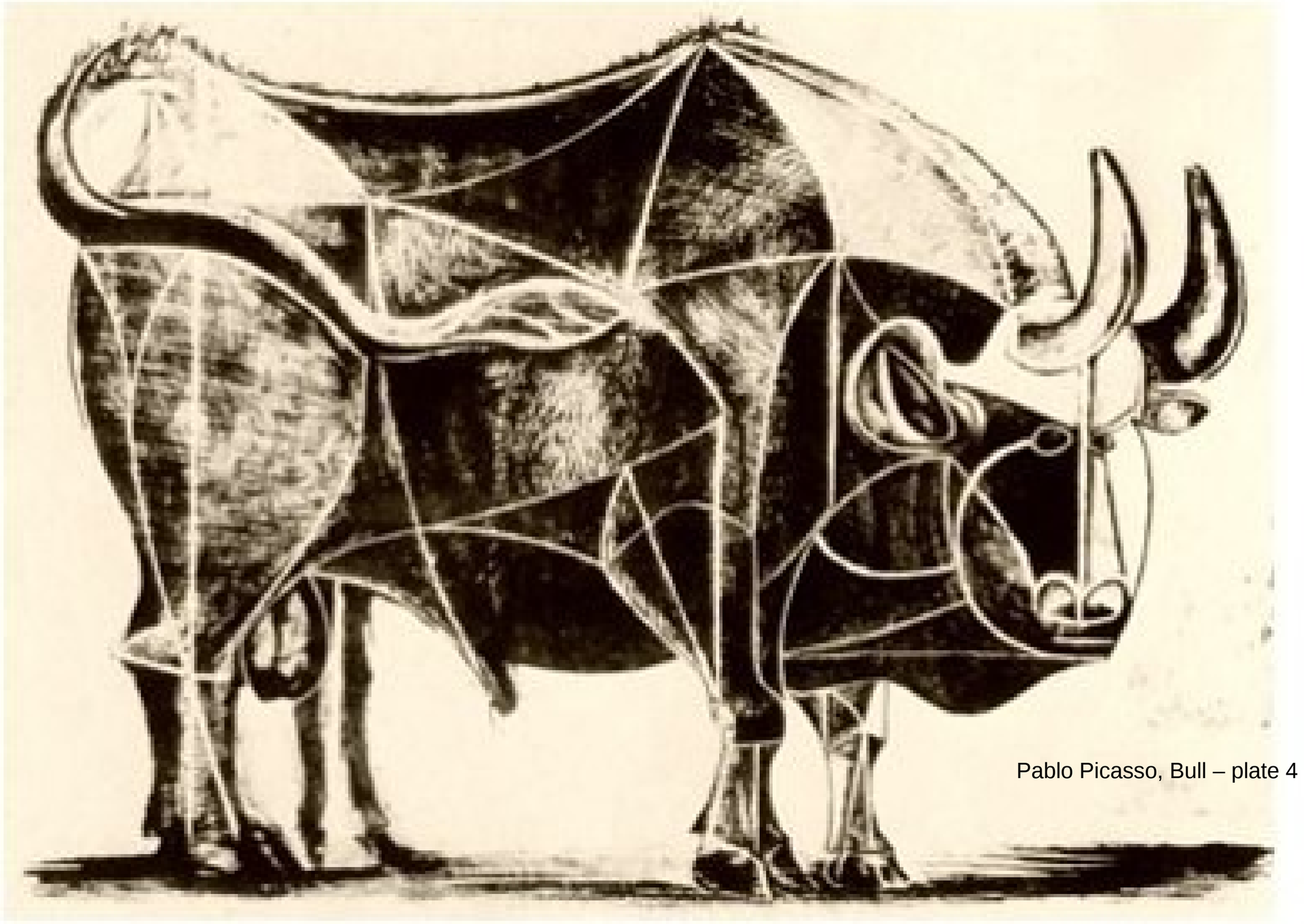
Pablo Picasso, Bull – plate 2

TOPCONF
Duesseldorf

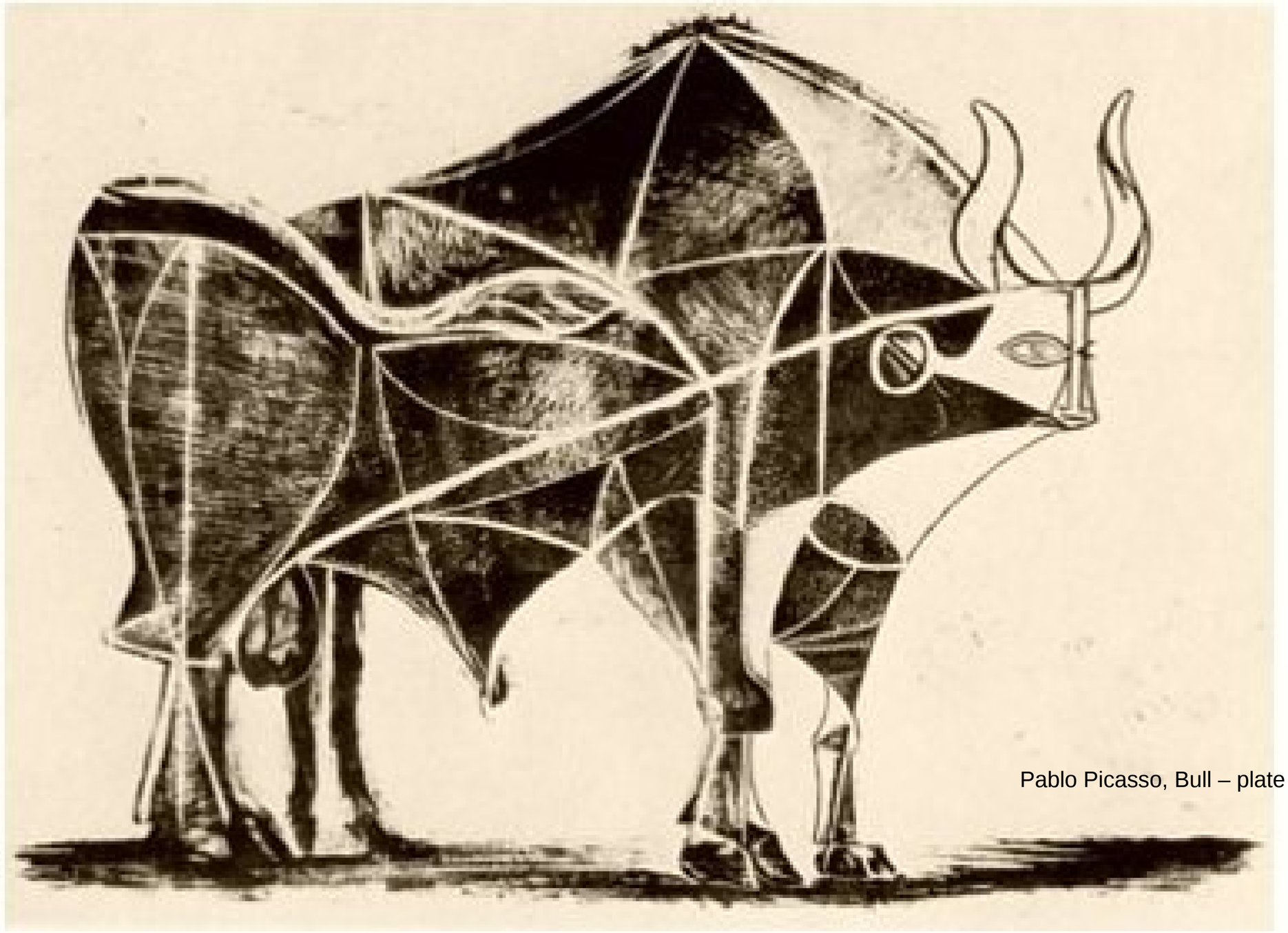


Pablo Picasso, Bull – plate 3

TOPCONF
Duesseldorf

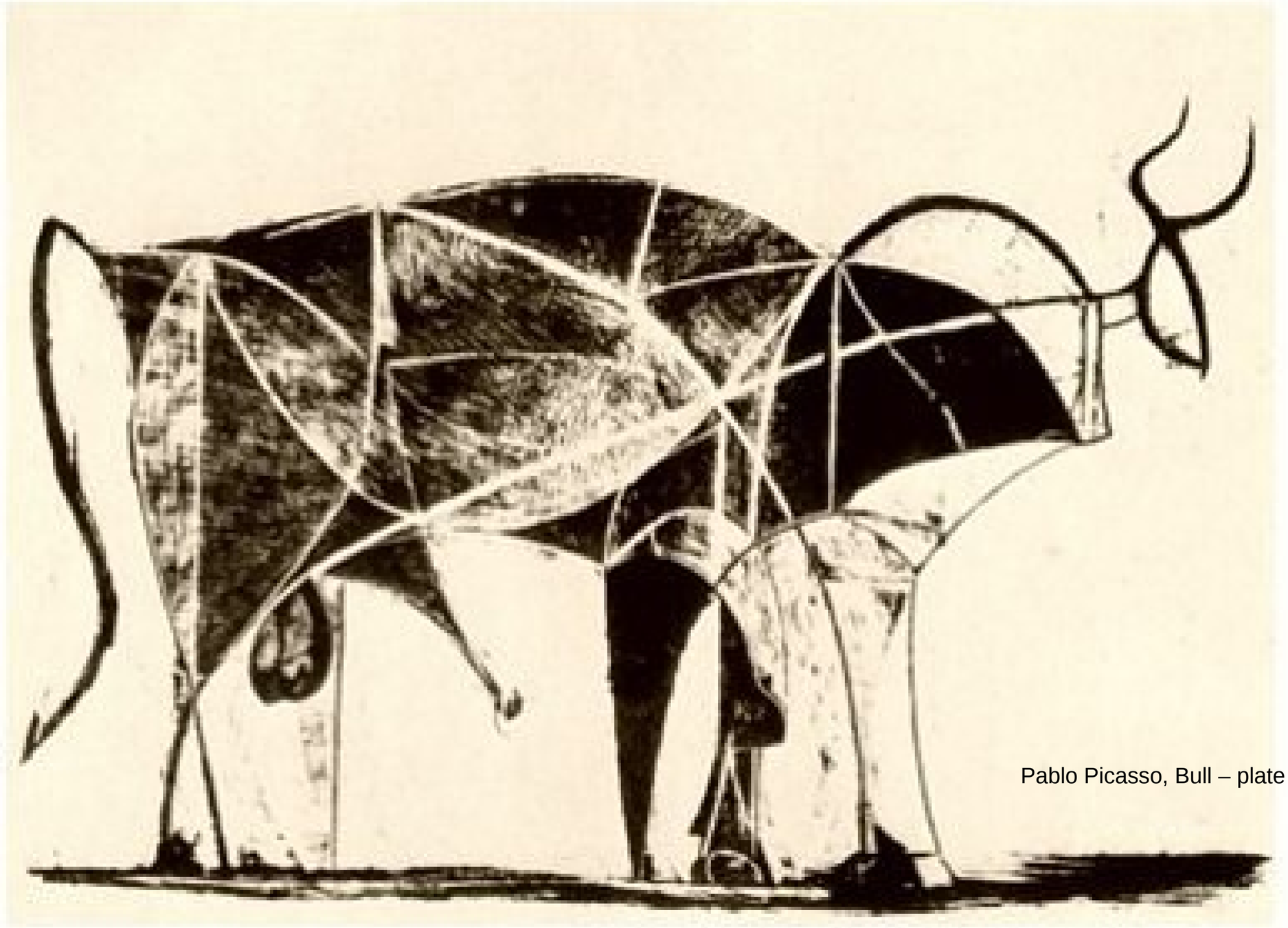


Pablo Picasso, Bull – plate 4

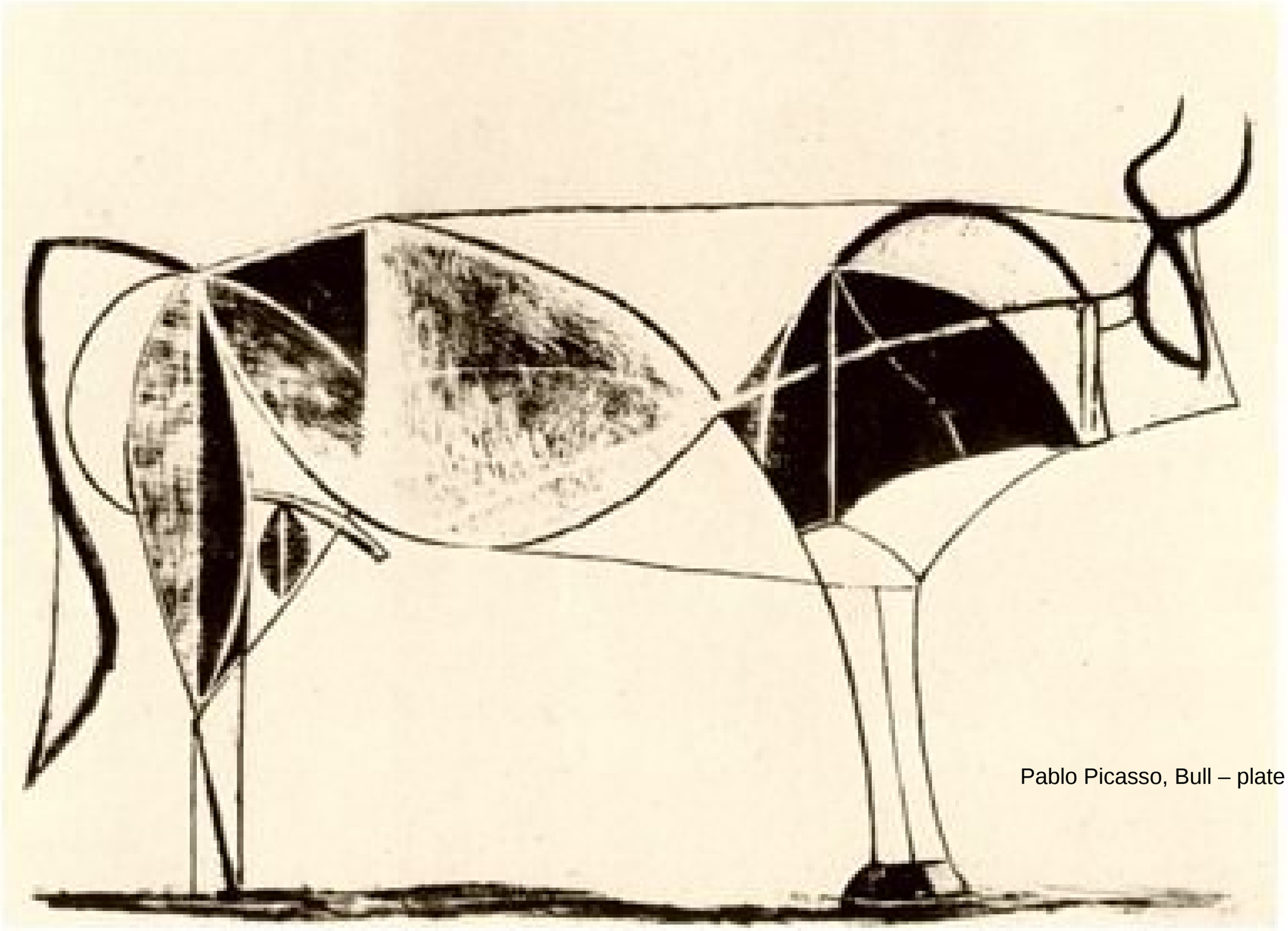


Pablo Picasso, Bull – plate 5

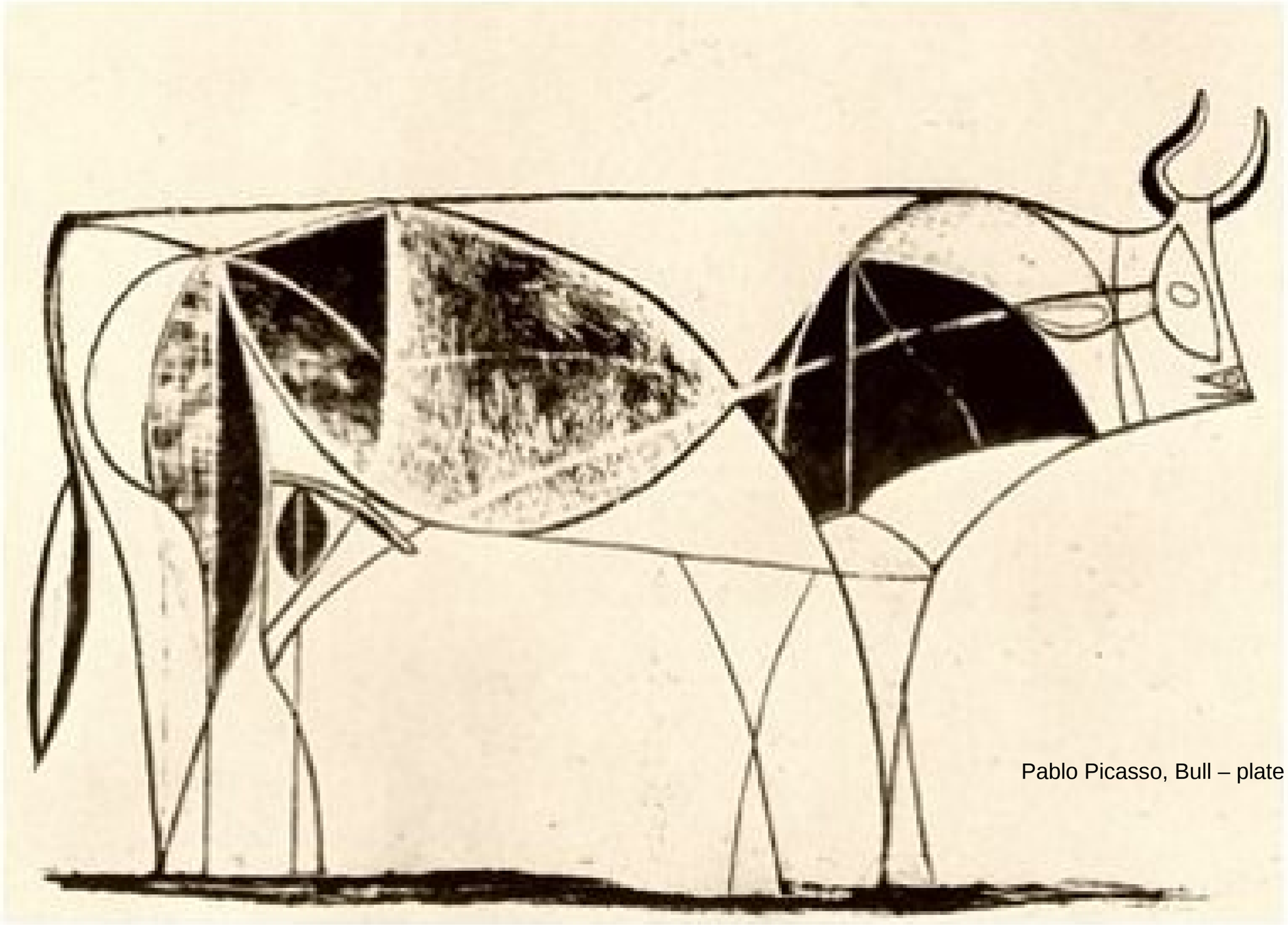
TOPCONF
Duesseldorf



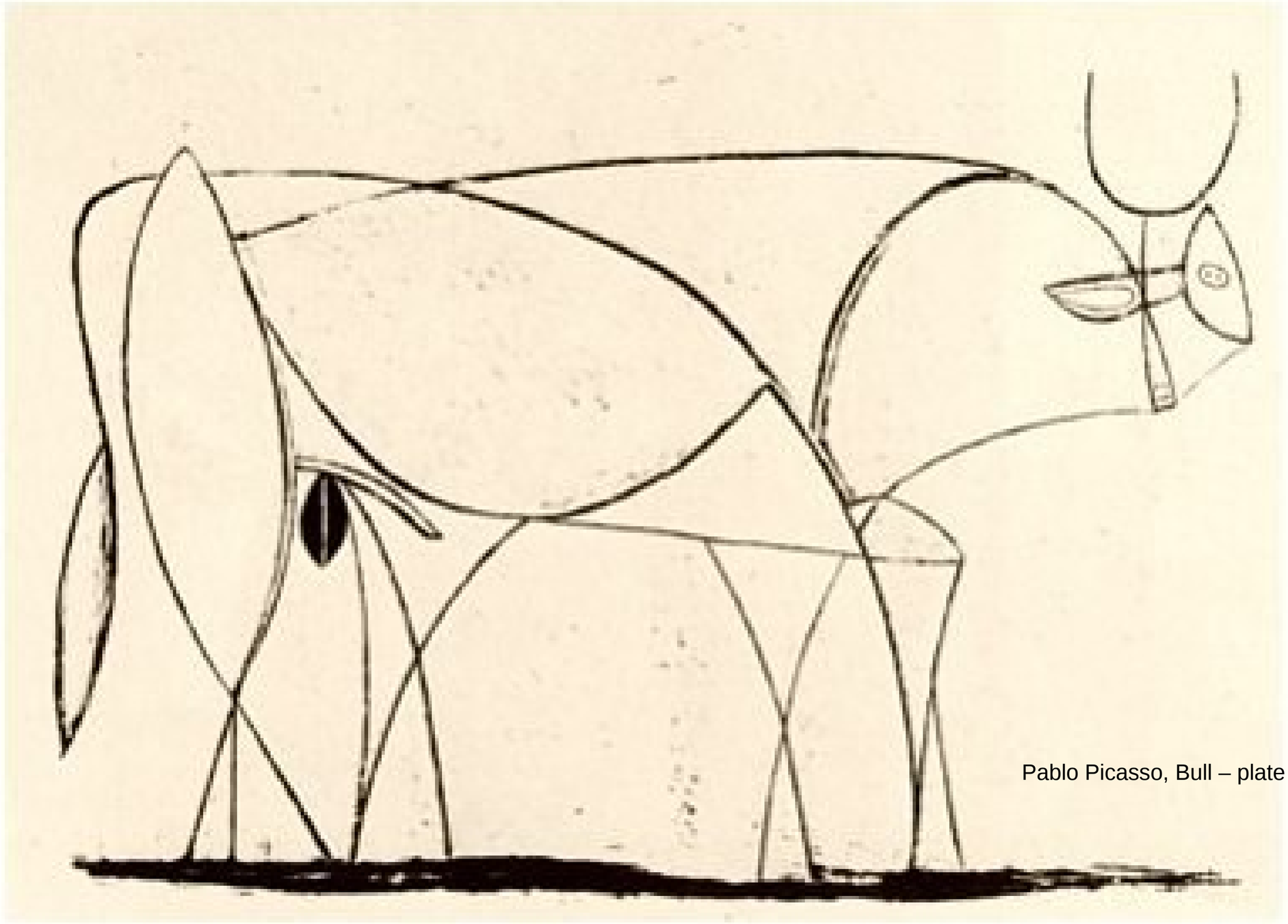
Pablo Picasso, Bull – plate 6



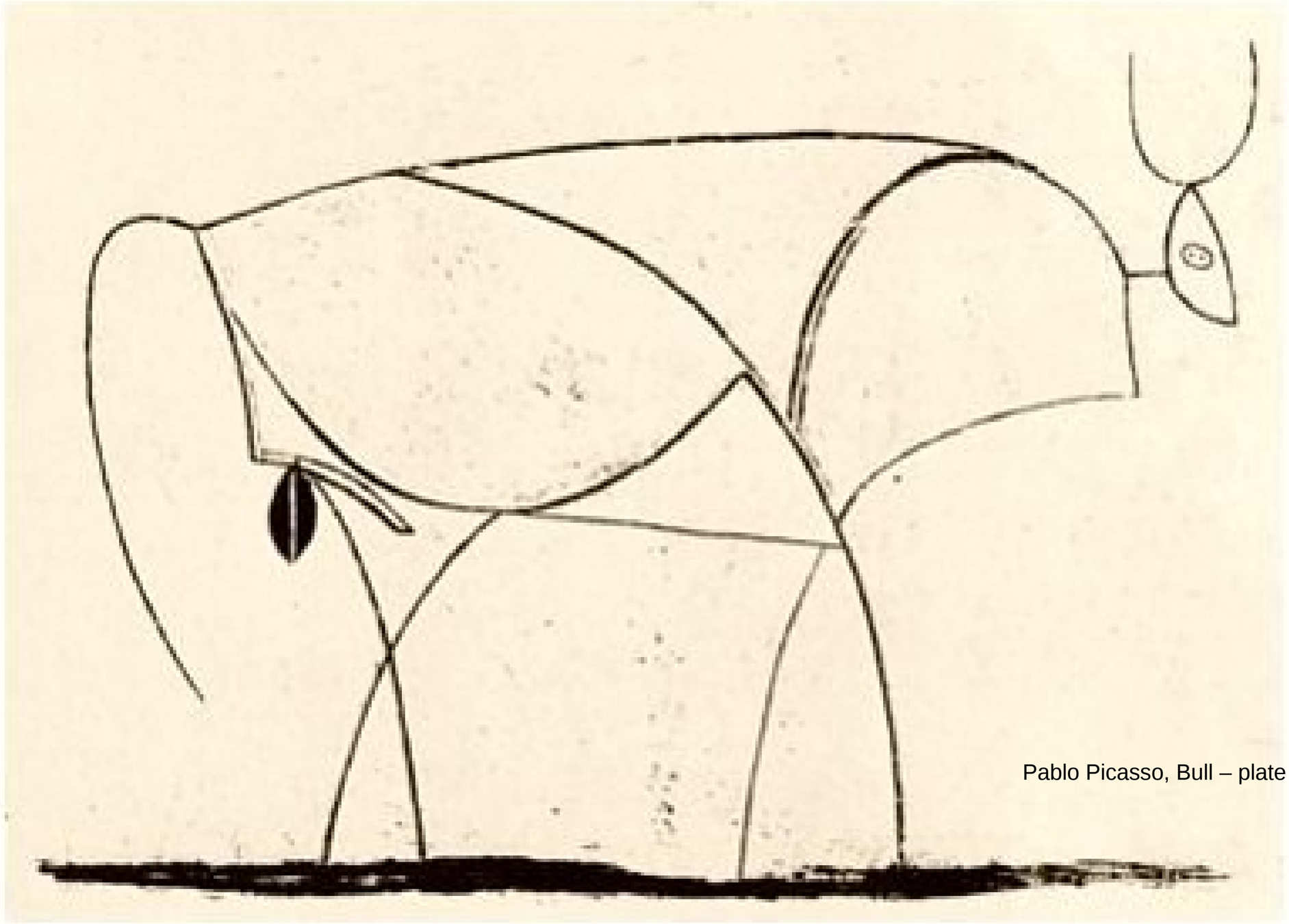
Pablo Picasso, Bull – plate 7



Pablo Picasso, Bull – plate 8

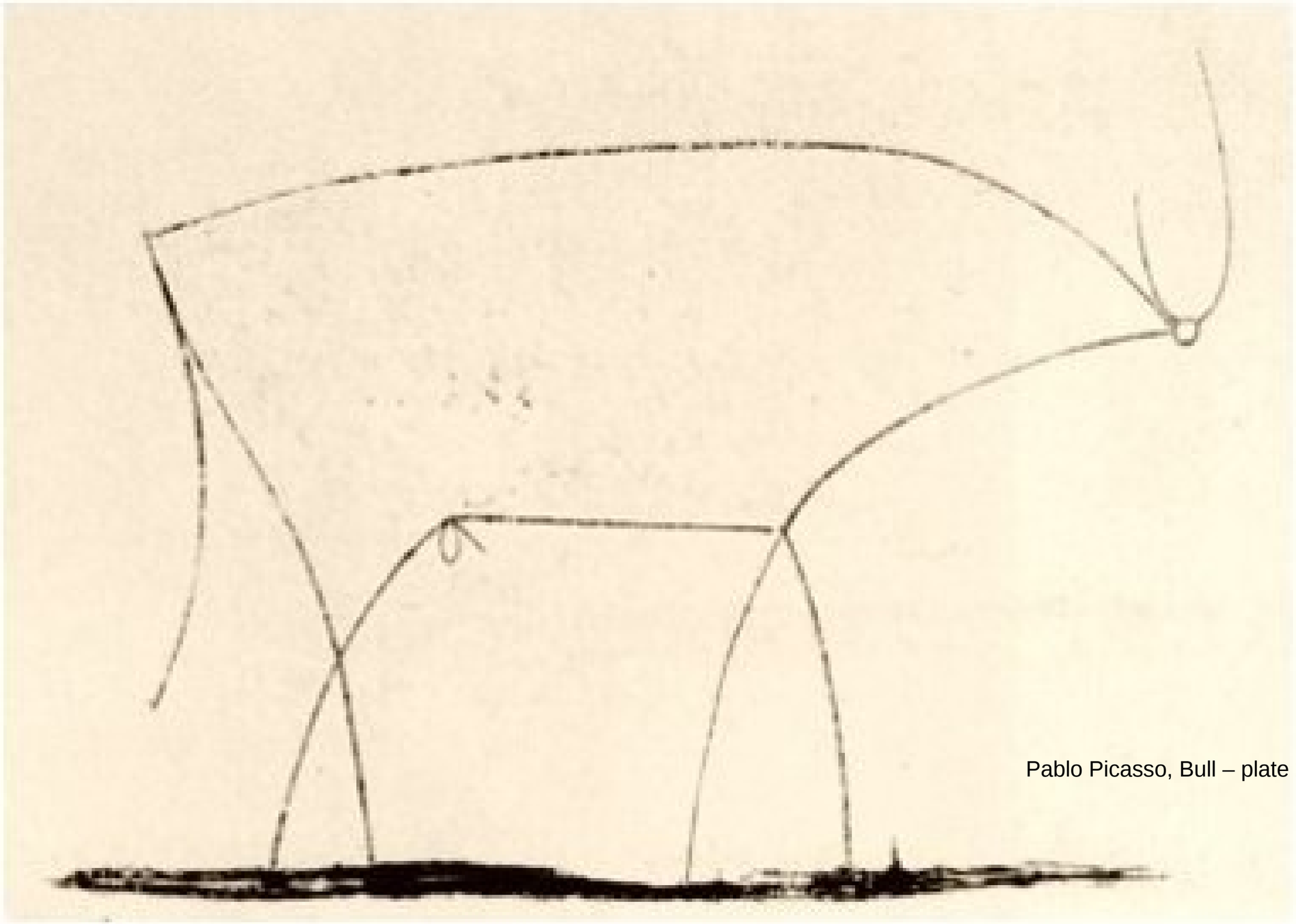


Pablo Picasso, Bull – plate 9



Pablo Picasso, Bull – plate 10

TOPCONF
Duesseldorf



Pablo Picasso, Bull – plate 11

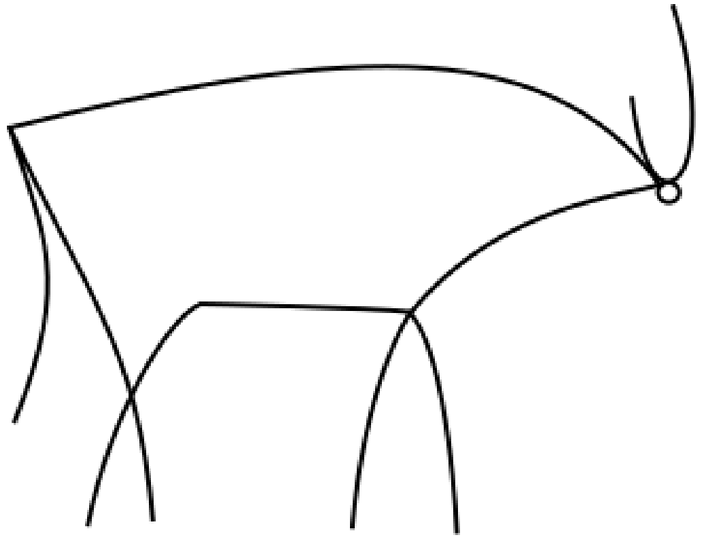
TOPCONF
Duesseldorf

Pepper:
Mmpfat?

Pablo Picasso, Bull – plate 11

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy



Abstract

Concrete

Greatest Common Divisor

156	180
2	2
2	2
3	3
13	3
	5

2*2*3 = 12

Greatest Common Divisor

156

2

2

3

11

180

2

2

3

3

5

} 2*2*3 = 12

Pepper:
Isn't factorizing expensive?

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy

Greatest Common Divisor in C

```
int gcd(int n, int m) {
  while (n != 0) {
    int t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

Modulo both numbers

Replace larger number by remainder

Repeat until zero

Return non-zero one

Greatest Common Divisor in C

```
int gcd(int n, int m) {
    while (n != 0) {
        int t = m % n;
        m = n;
        n = t;
    }
    return m;
}
```

Pepper:

What if I need the gcd of a `long` value?

Image from Pepper & Carrot by David Revoy

Greatest Common Divisor in C++

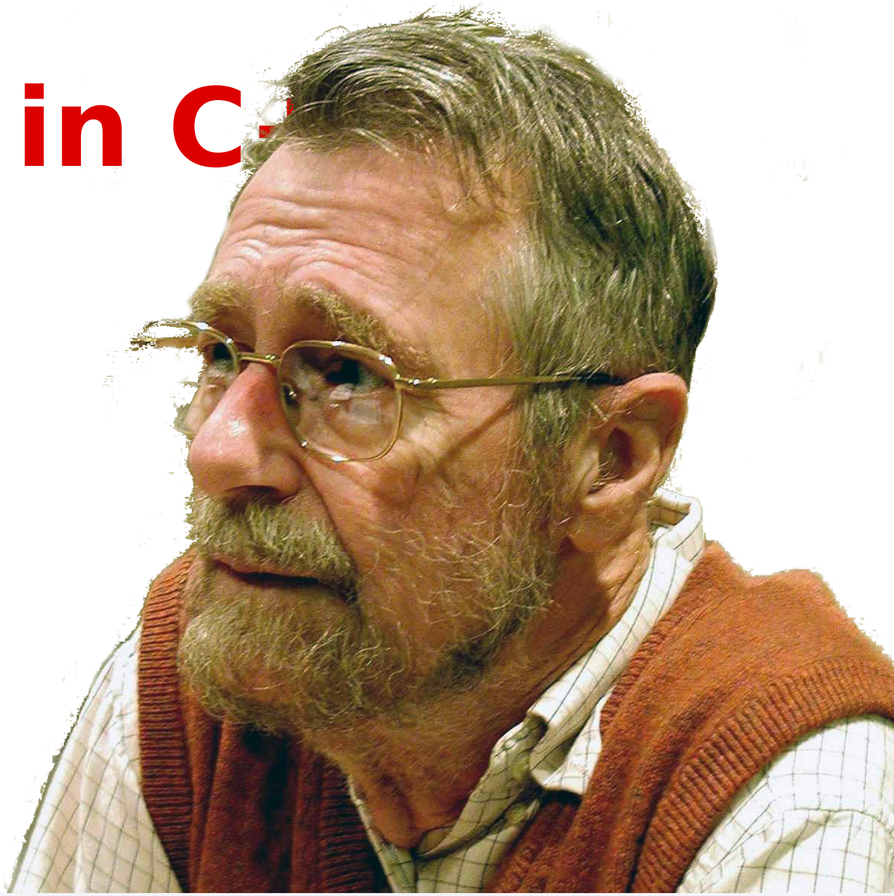
```
template <R>
R   gcd(R   m, R   n) {
  while (n != R(0)) {
    R   t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

Greatest Common Divisor in C++ish

```
template <EuclideanDomain R>
R   gcd(R   m, R   n) {
  while (n != R(0)) {
    R   t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

Greatest Common Divisor in C++

```
template <EuclideanDomain R>
R   gcd(R   m, R   n) {
  while (n != R(0)) {
    R   t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```



Edsger Wybe Dijkstra:
Being abstract is something profoundly different from being vague …
The purpose of abstraction is not to be vague,
but to create a new semantic level in which one can be absolutely precise.

image from Wikipedia

Greatest Common Divisor in D

D template definition

D template call

```
R gcd(R)(R m, R n) {
  static assert (isEuclideanDomain!R);
  while (n != R(0)) {
    R t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

Implementing isEuclideanDomain

```
enum bool isEuclideanDomain(R) =
    is(ReturnType!((R r) => r % r) == R)   &&
    is(ReturnType!((R r) => r != R(0)) == bool)
```

Custom types work with gcd

```
struct Euro {
    double amount;

    T opBinary(string op)(T rhs) {
        return mixin("amount "~op~" rhs.amount");
    }
    bool opEquals()(auto ref const S s) const {
        return amount == s.amount;
    }
}
```

overload all binary operators via string mixins and CTFE

overload == and !=

Greatest Common Divisor in D

```
R gcd(R)(R m, R n) if (isEuclideanDomain!R) {

  while (n != R(0)) {
    R t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

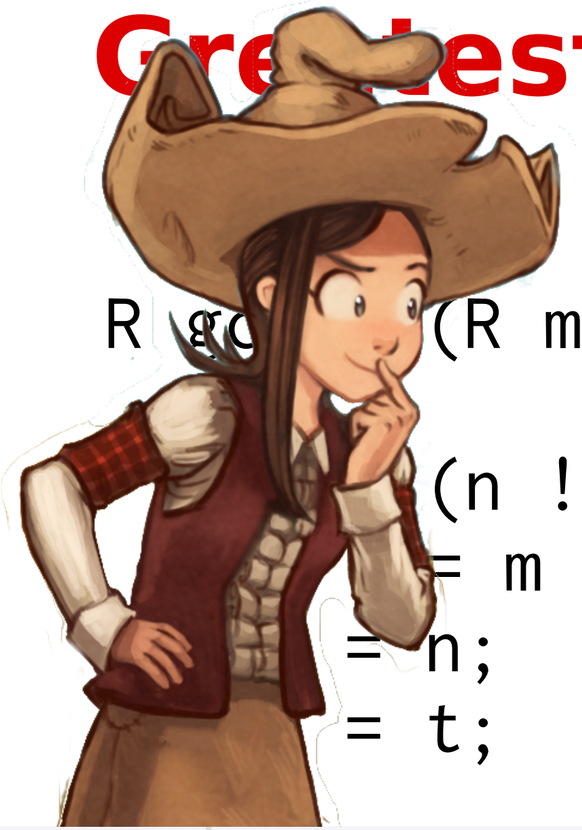
overload by static-if

Greatest Common Divisor in D

```
R gcd(R)(R m, R n) if (isEuclideanDomain!R
                       && !is(R : BigInteger) {
  while (n != R(0)) {
    R t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

Use another one
for BigInteger

Greatest Common Divisor in D



```
R gc(R m, R n) if (isEuclideanDomain!R
                   && !is(R : BigInteger) {
    (n != R(0)) {
     = m % n;
    = n;
    = t;
}
return m;
}
```

Use another one
for BigInteger

Pepper:
So if gcd is an abstraction, is it „leaky“?

Image from Pepper & Carrot by David Revoy

Gr t Common Divisor in D

```
R  R m, R n) if (isEuclideanDomain!R
                 && !is(R : BigInteger) {
    w  != R(0)) {
        % n;
    }
    return m;
}
```

Use another one
for BigInteger

Pepper:
You know, all abstractions are leaky, they say.

Image from Pepper & Carrot by David Revoy

“Leaky” Abstractions

Joel Spolsky:
All non-trivial abstractions, to some degree, are leaky.

“Leaky” Abstractions

Gregor Kiczales:
The "abstractions" we manipulate are not, in point of fact, abstract. They are backed by real pieces of code, running on real machines, consuming real energy, and taking up real space. [...] What is possible is to temporarily set aside concern for some (or even all) of the laws of physics.

Forget about “Leaky” Abstractions

Forget [illegible] “Leaky” Abstractions

Pepper:
Speaking about networking. I know a great joke about UDP,
but I'm not sure if you will get it.

Image from Pepper & Carrot by David Revoy

“Incomplete” or “Unsuitable”!

Incomplete can be fixed

Unsuitable is beyond repair

Allocators in C++

```
std::vector<int> v1;    // an integer array

// use something else than malloc
std::vector<int, tbb::scalable_allocator<T>> v2;
```

Do not use malloc

Custom Allocator in D

```
alias FList = FreeList!(GCAllocator, 0, unbounded);
alias A = Segregator!(
                8, FreeList!(GCAllocator, 0, 8),
              128, Bucketizer!(FList,    9,  128,  16),
              256, Bucketizer!(FList,  129,  256,  32),
              512, Bucketizer!(FList,  257,  512,  64),
             1024, Bucketizer!(FList,  513, 1024, 128),
             2048, Bucketizer!(FList, 1025, 2048, 256),
             3584, Bucketizer!(FList, 2049, 3584, 512),
    4072 * 1024, AllocatorList!(
                     () => BitmappedBlock!(GCAllocator, 4096)(4072 * 1024)),
    GCAllocator);
// example use:
A myAlloc;
auto b = myAlloc.allocate(500);
```

So many decisions

What alignment to use?
Can you choose alignment per allocation?
Is memory contiguous or not?
Is it thread-local or shared?
Can you deallocate?
Must you deallocate or is there a garbage collector?
Can you reallocate?
Can you find the start address of arbitrary pointers?

deallocate in Bucketizer

```
static if (hasMember!(Allocator, "deallocate"))
bool deallocate(void[] b)
{
    if (!b.ptr) return true;
    if (auto a = allocatorFor(b.length))
    {
        a.deallocate(b.ptr[0 .. goodAllocSize(b.length)]);
    }
    return true;
}
```

Only if sub-allocator has deallocate

Design by Introspection

Andrei Alexandrescu:
Each use of static if doubles the design space covered

image from Dconf 2015

Compile time introspection is useful

Allocators
Checked Integers
Serialization
Deep Copy
Struct of arrays
...

The power of D

Guillaume Piolat:
The combination of Static Introspection, generative features and CTFE forms the trinity that powers D's unique selling point.

image from https://github.com/p0nce

Lua Integration

```
print(sinus(42))
```

Want to execute this Lua script

Lua in C

```
static int l_sin (lua_State *L) {
  double d = luaL_checknumber(L, 1);
  double result = sin(d);
  lua_pushnumber(L, result);
  return 1; /* number of results */
}

lua_pushcfunction(l, l_sin);
lua_setglobal(l, "sinus");
```

Check parameter

Actual functionality

Convert result

Register function

Lua environment `l`

Lua in D

```
l["sinus"] = &sin;
```

L in

```
l[      = &sin;
```

Pepper:
Are you kidding me? Looks like cheating.

Image from Pepper & Carrot by David Revoy

```
l[       s"] = &sin;
```

Pepper:
Ooooh, it‘s magic? I love magic. Let it be magic, please.

Image from Pepper & Carrot by David Revoy

What happens inside the assignment syntactic sugar

```
this.push();
scope(success) lua_pop(this.state, 1);

pushValue(this.state, key);     // "sinus"
pushValue(this.state, value);   // &sin
lua_settable(this.state, -3);
```

Explicit state

Register function

What happens inside pushValue

```
void pushValue(T)(lua_State* L, T value)
{
  static if(is(T : LuaObject))
      value.push();
  else static if(is(T == LuaDynamic))
      value.object.push();
  // many more cases ...
  else static if(is(T : const(char)*))
      lua_pushstring(L, value);
  else static if(isSomeFunction!T)
      pushFunction(L, value);
  else
      static assert(false, "Unsupported type `" ~ T.stringof ~ "` in stack push
operation");
}
```

If-cascade for static type dispatch

Function case

What happens inside pushFunction

```
pushFunction(T)(lua_State* L, T func) if (isSomeFunction!T)
{
    lua_pushlightuserdata(L, func);
    lua_pushcclosure(L, &functionWrapper!T, 1);
}
```

Store func in closure

Automagic wrapping

What happens inside functionWrapper

```
extern(C) int functionWrapper(T)(lua_State* L)
{
    alias FillableParameterTypeTuple!T Args;
    enum requiredArgs = Args.length;
    int top = lua_gettop(L);
    if(top < requiredArgs)
        argsError(L, top, requiredArgs);
    T func = cast(T)lua_touserdata(L, lua_upvalueindex(1));
    Args args;  // a typed tuple where we put the parameters
    foreach(i, Arg; Args)
        args[i] = getArgument!(T, i)(L, i + 1);
    return callFunction!T(L, func, args);
}
```

Check arity

Get func from closure

Check argument types

Actual call

What happens inside callFunction

```
int callFunction(T)(lua_State* L, T func,
                    ParameterTypeTuple!T args)
{
    alias BindableReturnType!T RetType;
    RetType ret;
    try {
        ret = func(args);   // call &sin
    } catch (Exception e) {
        luaL_error(L, "%s", toStringz(e.toString()));
    }
    return pushReturnValues(L, ret);
}
```

Actual call

Another if-cascade

Lua in D

```
l["sinus"] = &sin;
```

Terry Pratchett:
It's still magic even if you know how it's done.

Read this talk as blog posts here

Andreas Zwinkau

KIT

beza1e1.tuxen.de

zwinkau@mailbox.org

TOPCONF
Duesseldorf

Read this talk as blog posts here

dreas Zwinkau

beza1e1.tuxen.de

zwinkau@mailbox.org

Pepper:
This is *magic*. You are all witches now.
k thx bye

TOPCONF Duesseldorf

Image from Pepper & Carrot by David Revoy

String interpolation as library

```
import scriptlike;
int num = 21;
writeln(mixin(interp!"The number ${num} doubled is ${num * 2}."));
// Output: The number 21 doubled is 42.
```